Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

FEW Phone File System

João Paulo da Conceição Soares nº 25940

1º Semestre de 2008/09
02 de Abril de 2009

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

FEW Phone File System

João Paulo da Conceição Soares nº 25940

Orientador: Prof. Doutor Nuno Manuel Ribeiro Preguiça

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

1º Semestre de 2008/09
02 de Abril de 2009

Acknowledgements

Gostaria de agradecer a todos aqueles que, directa ou indirectamente, contribuíram para a realização deste trabalho.

Este trabalho foi parcialmente suportado pelo Projecto Files EveryWhere (Project #POSC EIA 59064 2004) e por uma bolsa de investigação do Centro de Informática e Tecnologias da Informação da FCT-UNL

Em particular, gostava de agradecer ao meu orientador, Nuno Manuel Preguiça, não só pela disponibilidade, pelo apoio e pelos conselhos dados, mas acima de tudo pela grande dedicação e pelas oportunidades proporcionadas ao longo dos últimos anos, e ao longo da realização deste trabalho.

Gostaria também de agradecer aos meus colegas da 254, em especial ao Pedro Sousa, David Precatado e Nuno Luis pela disponibilidade e apoio que me deram, e acima de tudo pela paciência que tiveram para me aturar durante este período.

Ao professor João Lourenço, por todas as conversas e pela partilha de experiências vividas e que serviram de incentivo para terminar mais uma etapa da minha vida.

Aos meus pais e ao meu irmão, por todo o apoio e incentivo que me deram, não só na realização deste trabalho, mas durante todo o meu percurso académico, e que só a nossa família nos sabe dar.

Quero também deixar o meu agradecimento à Isabel, não só pelo incentivo e compreensão demonstrados, mas acima de tudo por fazer parte da minha vida.

Resumo

A evolução dos actuais telemóveis fez com que estes dispositivos deixassem de ser meros dispositivos de comunicação móvel. Actualmente é raro o telemóvel que não inclui uma câmara fotográfica digital, que não permite ler ficheiros multimédia (áudio e vídeo), ou que não possa ser utilizado como uma consola de jogos. É ainda mais raro o telemóvel que não ofereça suporte para aplicações desenvolvidas em Java, ou que não possua mais do que uma tecnologia de comunicação sem fios (e.g. GSM/GPRS, UMTS, Bluetooth e Wi-Fi).

Esta evolução tem sido possível graças aos avanços tecnológicos que permitiram, não só, um aumento da capacidade de computação destes dispositivos, bem como um aumento da capacidade de armazenamento e de comunicação dos mesmos.

Esta dissertação descrever um sistema de gestão de dados distribuído, baseado em replicação optimista, denominado FEW Phone File System. Este sistema tira partido da capacidade de armazenamento e de comunicação sem fios dos telemóveis actuais, possibilitando a um utilizador transportar, no seu telemóvel, os seus dados pessoais, permitindo aceder aos mesmos a partir de uma qualquer *workstation*, como se de ficheiros locais se tratassem.

O sistema descrito baseia-se num modelo híbrido que conjuga o modelo cliente-servidor com um modelo baseado em replicação *peer-to-peer*, recorrendo a reconciliação periódica para garantir a consistência entre réplicas. A aplicação servidor executa no telemóvel e o cliente na *workstation*. As comunicações entre cliente e servidor podem recorrer a múltiplas tecnologias de comunicação, permitindo ao FEW Phone File System adaptar-se dinamicamente às condições existentes.

Para lidar com as limitações de armazenamento e energia impostas pelo telemóvel, o sistema apresentado permite que conteúdos multimédia sejam adaptados às especificações deste dispositivo. Consegueo-se assim uma redução do volume de dados transferidos para o telemóvel, possibilitando armazenar maior volume de dados do utilizador. O FEW Phone File System integra também mecanismos que possibilitam manter informação sobre a existência de outras cópias dos ficheiros armazenados (e.g. WWW), evitando a sua transferência a partir do dispositivo móvel sempre que o acesso a essas cópias seja mais vantajoso. Dado que é actualmente

comum manter cópias de alguns conteúdos em diferentes servidores (e.g. CVS/SVN para programas, Picasa para fotografias), esta aproximação permite obter estes conteúdos a partir desses repositórios.

Palavras-chave:

- Sistema de replicação optimista de dados
- Reconciliação periódica
- Recodificação de dados
- Detecção de origem de dados
- Computação móvel

Abstract

The evolution of mobile phones has made these devices more than just simple mobile communication devices. Current mobile phones include such features as built-in digital cameras, the ability to play and record multimedia contents and also the possibility of playing games. Most of these devices have support for Java developed applications, as well as multiple wireless technologies (e.g. GSM/GPRS, UMTS, Bluetooth, and Wi-Fi).

All these features have been made possible due to technological evolution that led to the improvement of computational power, storage capacity, and communication capabilities of these devices.

This thesis presents a distributed data management system, based on optimistic replication, named FEW Phone File System. This system takes advantage of the storage capacity and wireless communication capabilities of current mobile phones, by allowing users to carry their personal data "in" their mobile phones, and to access it in any workstation, as if they were files in the local file system.

The FEW Phone File System is based on a hybrid architecture that merges the client/server model with peer-to-peer replication, that relies on periodic reconciliation to maintain consistency between replicas. The system's server side runs on the mobile phone, and the client on a workstation. The communication between the client and the server can be supported by one of multiple network technologies, allowing the FEW Phone File System to dynamically adapt to the available network connectivity.

The presented system addresses the mobile phone's storage and power limitations by allowing multimedia contents to be adapted to the device's specifications, thus reducing the volume of data transferred to the mobile phone, allowing for more user's data to be stored. The FEW Phone File System also integrates mechanisms that maintain information about the existence of other copies of the stored files (e.g. WWW), avoiding the transfer of those files from the mobile device whenever accessing those copies is advantageous. Due to the increasing number of on-line storage resources (e.g. CVS/SVN, Picasa), this approach allows for those resources to be used by the FEW Phone File System to obtain the stored copies of the user's files.

Keywords:

- Optimistic data replication system
- Periodic reconciliation
- Data transcoding
- Data sources detection
- Mobile computing

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1 Context

In the last few years, mobile phones have evolved from simple communication devices to powerful computing devices. Current mobile phones allow users to read multimedia files, take pictures with built-in digital cameras and even play games. This has been possible due to the evolution of mobile phones' computational capabilities and storage capacity. The communication capabilities of these devices have also evolved from wide-area, low bandwidth network technologies (GSM) to include some sort of local-area, high bandwidth network technologies such as Bluetooth or Wi-Fi. Additionally, popularity of these devices is huge, with almost everyone carrying at least one mobile phone with him most of the time [3].

The main goal of this work is to develop a distributed data management system, that will allow users to explore the available storage capacity of today's mobile phones, enabling them to access their files, at any moment and in any place. The FEW Phone File System is designed to allow users to carry personal data "in" their mobile phones and provide easy access to it, as normal data, from any nearby computer with a local area network card (e.g. Bluetooth, Wi-Fi).

1.2 Motivation

In a world with a growing trend for mobile and pervasive computing, the availability of data is of great importance. Currently most users rely on network connectivity to servers or use personal storage devices to access their personal data. This gives users the possibility of accessing their personal data independently of location but at a cost. The following sections will introduce some of the most common methods employed by users to access their personal data independently of their location, as well as their properties.

1.2.1 Distributed File Systems

Distributed file systems [21, 4] have been developed with the general purpose of allowing users to access their personal data in any computer with network connectivity and a client agent present, thus decoupling users from a particular host.

The main drawback of distributed file systems is related to their design. While these systems can offer performances that match local file systems, these can only be obtain in the presence of high speed and low latency communication infrastructures. This is the reason why most of these systems have been developed for use in a local area network (LAN). When only low bandwidth and high latency networks are available, the performance of these systems tends to be unacceptable for regular users.

Another problem of distributed file systems is service availability. A distributed file system might fail to work in situations where network connectivity is unavailable, in face of server failure (e.g. NFS [21]) or if the client software is not present. Some distributed file systems (e.g. Coda [14, 26], Ficus [23, 8, 10]) alleviate these problems by providing support for disconnected operations [12]. However, in any case, these solutions always require a server infrastructure to exist, which can be problematic for some users and Internet settings (e.g. due to firewalls).

1.2.2 Portable Storage Devices

Due to the problems of distributed file systems, and their relative small availability, most people adopt portable storage devices as a means of storing and accessing their personal data independently of their location.

The drawbacks of such devices are related mostly to robustness and management issues. If a user loses or someone steals his portable storage device (PSD) or if that device becomes defective, the user will lose its personal data.

In order to avoid these problems, a user must periodically synchronize the PSD data with a personal workstation or a home PC. The management problem is even worse because users tend to have multiple PSD and keep replicas of their data in all of them. This puts much stress on a user, forcing him to deal with multiple file replicas.

A similar problem arises when a user forgets to bring the storage device with him. If the user's current workstation has an up-to-date copy of the data, then the user can access it to produce his work, still leaving him with the task of later synchronizing the data to the PSD. When the user's current workstation has an old version of the data, the user may be forced to produce concurrent versions of his work. If the current workstation has no version at all, he will be unable to produce any work.

1.2.3 Internet-based solutions

Another alternative with growing popularity in our days are personal/shared Internet workspaces (e.g. Google Docs [5]). These solutions, like DFS, offer good performance only when a high performance network connection is present. When only a low bandwidth infrastructure is available or when there is no network connectivity at all, these systems become unusable.

Furthermore, these solutions rely on third party providers, leaving the users with security and privacy problems to deal with.

1.3 General Overview and Contributions

This thesis presents the design, implementation and evaluation of a distributed data management system for mobile computers. This system includes a set of solutions to address the problem of personal data access on mobile environments.

These solutions and strategies are designed to address the problems created by mobile computing environments, including the limitations imposed by mobile devices that characterize our system. The following sections will briefly describe the system.

1.3.1 FEW Phone File System

The FEW Phone File System (FEW Phone FS) is a distributed data management system for mobile environments, relying on optimistic replication. It is designed to allow users to carry their personal data on their mobile phones, and access it independently of their location. This is accomplished by maintaining on the mobile phone's storage system the most up-to-date version of the users' data, and allowing for replicas to be created and accessed whenever needed.

The use of a mobile device, such as a mobile phone, imposes some limitations to the system's design, namely computational, storage and energy limitations, that need to be considered in the design of the FEW Phone FS.

Optimistic replication allows for any host containing a replica of the data to access it at any time, without any synchronization with other nodes. This minimizes network communications, and connectivity requirements for performing updates, which is important due to the limitations imposed by the mobile phone. Since optimistic replication systems allow for disconnected operations, the FEW Phone FS relies on a periodic reconciliation process to maintain consistency

among replicas. This process is also responsible for automatically detecting and resolving possible conflicts, that arise from concurrent updates.

The FEW Phone FS is based on a hybrid architecture that merges the client/server model, relying on the mobile phone as a central server, with peer-to-peer replication. This allows the mobile phone to provide data access at any time. However, when it is more efficient to obtain a resource from a remote site rather than using a wireless connection to the mobile device, it is possible to rely on peer-to-peer communications. This minimizes the mobile device's energy consumption, while maintaining or even improving the system's performance. For these reasons, this presents as a good model for a system with the FEW Phone FS's requirements, since it addresses the limitations imposed by the mobile phone.

The communication between the client and the server can be supported by one of the multiple network technologies available. The FEW Phone FS dynamically adapts to the available network connectivity, in order for the client and server to communicate.

In order for the FEW Phone FS to address the storage limitations imposed by the usage of a mobile device, it allows for the multimedia contents to be adapted to the device's specifications. This results in the reduction of multimedia data stored on the mobile phone, allowing for more contents to be stored. Since less data is transferred to/from the mobile device, it also results in a reduction of power consumption and an increase in performance.

With these goals in mind, the FEW Phone FS also integrates a mechanism to detect the source of the contents stored on the system. This way, if the contents have been transferred from a remote site (e.g. WWW), the FEW Phone FS can try to use that site's URL to obtain the resource, rather than transferring it from the mobile phone during the reconciliation process. This mechanism is also used to allow users to access the full fidelity version of their data, by obtaining the replicas stored in other machines of the system.

These mechanisms are built to deal with the storage and power limitations imposed by the mobile phone, by reducing the volume of data stored and transferred to/from the mobile phone.

1.3.2 Contributions

As explained, the FEW Phone FS is designed to deal with the characteristics of mobile computing environments, as well as the limitations imposed by mobile devices. The most important aspects of its design are the following:

- An optimistic replication mechanism, that allows for replicas to be created and accessed

when needed. Due to the characteristics of this model, it also allows for disconnected operation.

- Reconciliation techniques that automatically detect and resolve possible conflicting updates in the file system.
- Mechanisms that allow for multimedia contents to be adapted to the mobile device's specifications, thus reducing storage and communication overhead.
- Mechanisms for identifying the sources of the data stored in the system, allowing access any of the existing copies of that data based on that information, and the state of the mobile device's power resources and connectivity.

1.4 Outline

The remainder of this document is organized as follows: Chapter 2 describes the design and architecture of the FEW Phone File System, and is organized into two subsections. Section 2.1 addresses the design requirements and the goals for this project, while Section 2.2 describes the architecture of the system. Chapter 3 describes the implementation details of the prototype for the FEW Phone File System. Chapter 4 presents the evaluation of the system. Chapter 5 discusses related work. Chapter 6 presents the final remarks of this work as well as some possible improvements for the future.

2. Design

Most distributed file systems [21, 4] are designed to be used in high performance local network areas. Some systems also support mobile use (e.g. Footloose [19]).

The FEW Phone File System (FEW Phone FS) is a distributed data management system, designed to allow users to access their personal files as regular files in a local file system, independently of the machine's location or network connectivity.

To achieve this, the FEW Phone FS takes advantage of a mobile device that most people carry with them at all times (sometimes more than one): the mobile phone.

Most of today's mobile phones feature significant amounts of storage space, ranging from few hundreds of megabytes to many gigabytes of memory, and also feature many different wireless communication capabilities, such as: GSM/GPRS/UMTS, Bluetooth, or even Wi-Fi.

These characteristics make these devices ideal for supporting the FEW Phone FS, by allowing to store the necessary data and easily contacting nearby computers. Thus the mobile phone is the central component of the FEW Phone FS.

The next section addresses the requirements and goals for the system.

2.1 Requirements and Goals

The goal of this work is to develop a distributed data management system that allows users to access their personal data independently of their location and network connectivity. To achieve this, the FEW Phone FS takes advantage of the storage capacity and wireless communication capabilities of current mobile phones.

The FEW Phone FS allows users to carry, on their mobile phones, their personal files, and access them as regular files in a file system, as depicted in Figure 2.1. Due to the mobile nature of the device in question, this can be made possible in any machine, independently of the user's location. This is done by starting the FEW Phone FS client in any computer, and relying on the mobile phone to act as a centralized file server for the FEW Phone FS system.

For users to access the data stored on the mobile phone's memory, communications between the client (desktop computer) and the server (mobile phone) rely on any available wireless technology (e.g. Bluetooth, Wi-Fi). This way, file requests made to the client may be redirected

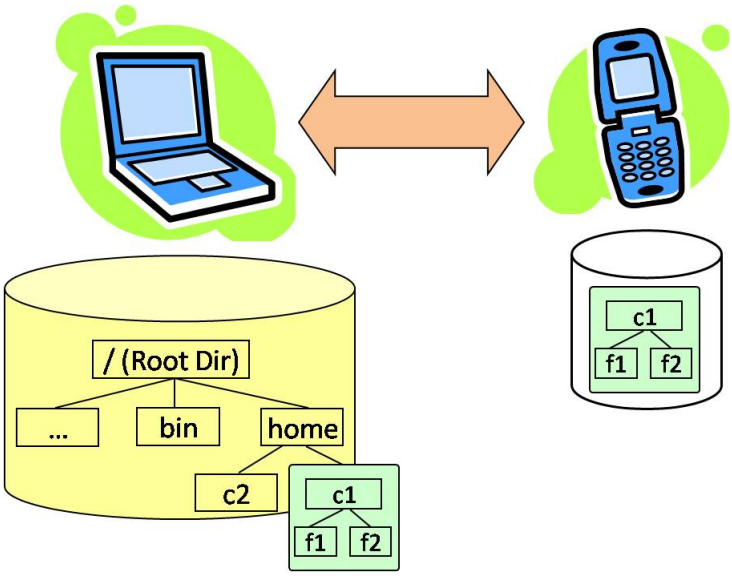

Figure 2.1 The FEW Phone File System.

to the server. The major drawbacks of these network technologies, when compared to high-speed wired networks, are lower bandwidth and higher latencies. To overcome these problems it is essential to design the FEW Phone FS as a replicated file system, rather than a remote file system.

Distributed file systems [21, 4] allow for short lived replicas of the server's files to be automatically created on the client's main memory (a few systems also store these replicas on disk [14, 27]). These replicas are normally created on a first-request basis, and stored for short periods of time, usually not surviving reboots. This way, instead of relying solely on the server to answer all requests made to the file system, these requests can be served from the client's cache, whenever possible. Performance is enhanced due to the reduction of the latency overhead imposed by client-server communications.

Replication-based systems [9, 8, 23] extend this design by creating long lived replicas on the node's local file system. This is accomplished by creating replicas of the entire or partial contents of the system. These systems can be split into two categories: pessimistic and optimistic replication systems.

Pessimistic replication systems still rely on server communication before accessing replicas, thus imposing an important overhead. On the other hand, optimistic replication allows for updates to be made directly on the local replicas, thus reducing communications. Since accessing

and updating replicas does not rely on the availability of the server, optimistic replication also allows for disconnected operations [14, 9]. Therefore, optimistic replication presents itself as a good model for the FEW Phone FS, allowing for replicas to be created in any desktop computer.

Since the major goal for the FEW Phone FS is to allow users to access their personal data, the system must be designed so that it can be used with a directory already containing personal data, thus freeing users from additional setup operations. The FEW Phone FS must then automatically and transparently replicate the user's data into the mobile phone's file system. The mobile phone will then be used as a server for the replication system, since it will act as a portable "central" data repository, allowing users to access their personal data from any computer by using the information stored in the mobile phone.

As any replicated system, the FEW Phone FS must be able to deal with data updates performed by the user. The system must propagate these updates to all replicas in order to maintain data consistency. It is, therefore, essential to design a synchronization process that automatically propagates updates among replicas.

The synchronization process must be able to propagate updates between replicas stored on the local machine and on the mobile phone. This way, the mobile phone will tend to store the most recent version of each file, allowing users to access their up-to-date data even when working in disconnected machines.

As said before, optimistic replication systems allow for disconnected operations. This increases the possibility for the occurrence of conflicting updates [7, 24]. For this reason, the synchronization process must also be able to, automatically, detect and resolve conflicts, thus minimizing the user's intervention.

Since this system is designed for a mobile device, there are several aspects that the FEW Phone FS must deal with. The two major aspects that need to be addressed are power consumption and storage capacity limitations.

Power consumption, on mobile devices, is mainly related to: CPU usage, and network operations, while storage limitation, on the other hand, is directly related to the volume of data stored on the mobile phone's memory.

To reduce power consumption, it is essential for the FEW Phone FS to reduce the number of tasks performed on the mobile phone. This can be accomplished by executing tasks, as much as possible, on fixed nodes, rather than on the mobile phone. Also, the number of network operations can be reduced, and the volume of data transferred during these operations minimized, thus reducing the mobile phone's power consumption.

Storage limitations may only be addressed by reducing the volume of data written to the mobile phone's memory.

All these facts must be taken into consideration when designing the FEW Phone FS. The integration of a *Data Transcoding* (DTC) module and a *Data Source Verification* (DSV) module are essential to accomplish these objectives. These modules are described in the following section.

2.1.1 Data Transcoding & Data Source Verification Modules

Most current mobile phones have built-in high resolution digital cameras and are able to reproduce digital multimedia files, such as audio or video. This leads users to store a considerable amount of multimedia contents in their mobile phones. However these devices usually cannot reproduce the contents with the maximum quality. For example, a 2 mega pixels digital photograph with 32 bits color depth, can only be presented with a resolution of 320x240 and a color depth of 16 bit, in most high-end mobile phones.

The DTC module is designed to convert multimedia contents to best fit the specifications of the mobile phone. The 2 mega-pixel photograph could be rescaled and its color depth reduced in order to best fit the mobile phone's screen specifications. This will allow for the size of multimedia contents to be reduced, which will lead to less storage consumption, as well as to the reduction of the data transferred to and from the mobile phone. Both these features will allow the FEW Phone FS to reduce the mobile phone's power consumption.

Due to the increasing number of on-line storage resources, more and more users rely on these systems for storing personal data or sharing data with other users. Take for example WWW file hosting systems such as RapidShare, MegaUpload, Picasa [6], or other data repository system such as CVS/SVN. These systems can be used as alternative sources for users to access their data.

The DSV module is designed to check and maintain the source of a file. If the contents of a new file have been transferred from a remote site (e.g. WWW), this module can be used to obtain that site's URL. This way, by keeping the remote sources for data items, a well connected machine (i.e. with a high-speed network connection) will be able to choose the most convenient source for retrieving data. For example, it may choose to transfer the data contents directly from the remote site, rather than transferring it from the mobile phone. This allows, not only for the mobile phone's power consumption to be reduced, by not transferring the replica's contents,

but also for the system's performance to be enhanced, due to a higher available bandwidth connection.

The two modules work together for the following goal: the reduction of data written and transferred into or from the mobile phone, while still allowing the client to access full-fidelity data, obtained from a remote replica in the system (or other remote site).

Less data written to the mobile phone's memory not only allows for more user information to be stored, but also for the reduction of power consumption. Power consumption will also be reduced since the communication channels transfer less amounts of data.

Combining the two modules also enhances the overall performance of the system. This way, a well connected machine will be able to transfer the data contents directly from remote sites, in peer-to-peer interaction, not depending on the low bandwidth connectivity to the mobile phone.

The next sections of this chapter will describe the architecture of the FEW Phone File System and its modules.

2.2 Architecture

As mentioned earlier, the FEW Phone FS is a distributed data management system for mobile environments, relying on optimistic replication. It is designed to allow users to carry, on their mobile phones, replicas of their personal files and access them as regular files in the file system of any computer.

Figure 2.2 presents the different components of the FEW Phone FS. Unlike the common approach, the mobile phone acts as a (portable) server in our system, as it maintains a replica of the data. The server architecture is simpler than the client's due to the limitations imposed by the mobile phone. The FEW Phone FS is built so that the more complex operations and computations remain on the client side.

The FEW Phone FS is based on optimistic replication, allowing for replicas to be created in any node whenever needed, and after that, accessed without the need for server interaction. This architecture allows full user access to any replica without the need of communicating with the mobile phone, while increasing the overall performance of the system. The reduction of communications between the mobile phone and clients is also highly important due to the limitations imposed by mobile devices. The performance increase leads to a better usability.

The FEW Phone FS is designed as a hybrid system because it uses a central server, but it

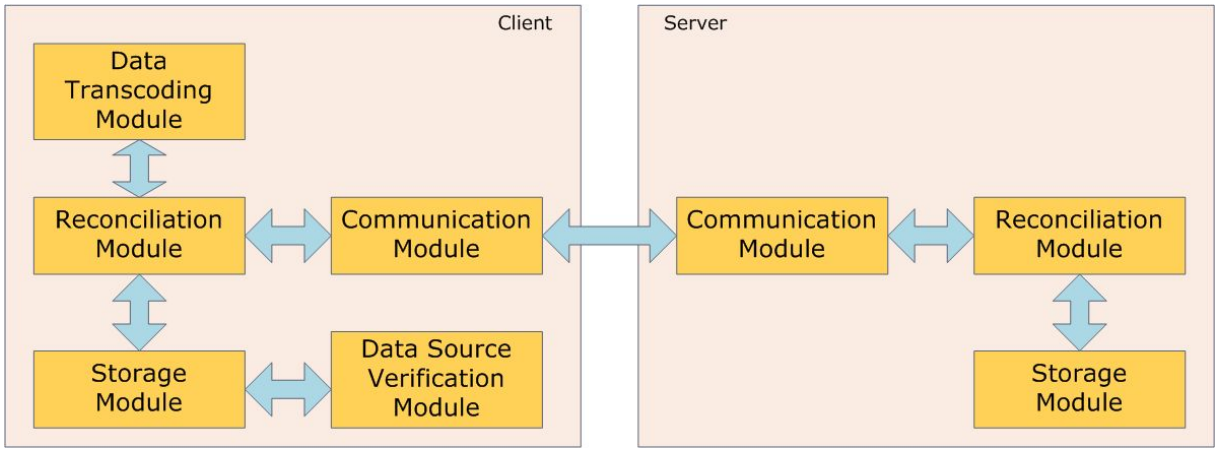

Figure 2.2 FEW Phone File System's design.

also uses peer-to-peer interaction. The mobile phone acts as the central server for the system, storing the most up-to-date version of each replica. Due to the already mentioned limitations of such devices, the FEW Phone FS may also use peer-to-peer communication when possible in order to reduce the mobile phone's power consumption. This model presents a valid option since it extends the "life" of the mobile phone, and also maintains, or even enhances, the overall performance of the system. Peer-to-peer replication will be presented in more detail later.

The FEW Phone FS operates on sets of files known as *containers*. A container is a subset of the file system tree that can be seen as a file system that has its root on some base directory. This way, the replication mechanism has well defined boundaries, thus simplifying management tasks such as the definition of the set of files to replicate and to synchronize, allowing for this process to be initiated once for the whole volume. These containers are managed and maintained by the *Storage Module*. This module will be presented in Section 2.2.2.

In order to maintain replicas synchronized, the FEW Phone FS does not attempt instant update propagation. This would generate excessive communication with the mobile phone, since every single update made to a replica would force the client to access the server to propagate that update. Instead, consistency is maintained by a periodic *reconciliation* process. Periodic reconciliation allows for multiple updates to the same replica to be batched, and transmitted as a single update. This allows for performance gains due to the batch performance improvement, and also reduces communications, by avoiding overlapped updates from ever being sent. The reduction of communications is especially important due to the energy limitations of the mobile

phone.

The reconciliation process typically operates at the container granularity, thus synchronizing an entire container, but it is possible to synchronize only a subset of the file system. This process is described in detail in Section 2.2.3. The FEW Phone FS relies on the mobile phone to keep all replicas synchronized, since reconciliation always includes the mobile phone, thus allowing the system to maintain consistency even is disconnected machines. This is similar to the solution proposed in Footloose [19].

Due to the energy and storage limitations imposed by the mobile phone, the FEW Phone FS features a DTC module, and a DSV module, as described in Section 2.1.1.

The DTC module allows the system to deal with multimedia contents, by adapting these contents to the configuration defined in the mobile phone, thus reducing the volume of data to be transferred to, and stored on the mobile phone. This also allows the FEW Phone FS to deal with some of the Input/Output limitations of the mobile phone.

The DSV module allows the system to determine the source of the data stored in the file system. To this end, this module intercepts network communications to determine the origin of contents retrieved from remote sites. If a replica's contents has its source in a remote site (e.g. WWW) the FEW Phone FS keeps the URL for that site. This URL will then be used to transfer the original contents of the file from the remote site, avoiding, this way, transferring the contents of the replica into or from the mobile phone.

This is also used for maintaining information about all replicas in the system. For each replica stored on the server, a list of available sources, i.e. a list of all existing replicas of that file, is also maintained. This allows the system to retrieve replica contents relying on a peer-to-peer communication model, thus reducing the communication with the mobile phone.

By combining the DTC and the DSV modules, it is possible to allow users to access the original contents of a file. The FEW Phone FS is responsible for automatically retrieving the contents from a site that maintains it. This is accomplished by maintaining on a replica's meta-data the URLs of the site or sites (FEW Phone FS's nodes or WWW sites) that maintain the original contents of that object. This way, availability will be improved as the number of replicas increases.

The use of both of these modules, not only allows for the reduction of the mobile phone's power consumption, but it can also improve the performance of the system, as shown by the results presented in Chapter 4.

The following sections will describe, in more detail, the components that compose the system. The components of the client are represented in Figure 2.3. The components of the server are a subset of these components, as described earlier.

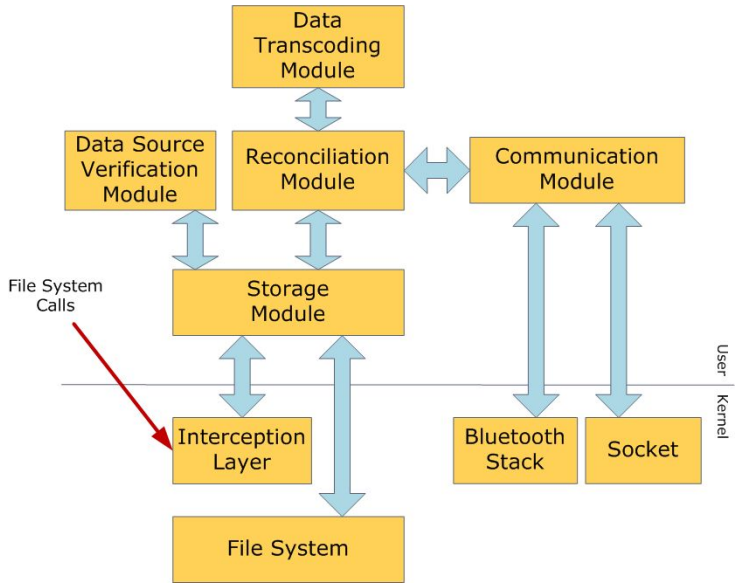

Figure 2.3 FEW Phone FS client's architecture.

Section 2.2.1 describes the architecture of the Communication module. Section 2.2.2 describes the architecture of the Storage module. Section 2.2.3 describe the architecture of the Reconciliation Module and the reconciliation process. Section 2.2.4 describes the Data Transcoding module. This section end with the description of the Data Source Verification module. The architectural differences between client and server will be presented whenever they exist.

2.2.1 Communication Module

This section describes the architecture of the Communication module. This module is responsible for creating the communication channel between the client (personal computer) and the server (mobile device), and among clients when using peer-to-peer communication.

The Communication module is designed to take advantage of the multiple available connectivity technologies. It is a modular component that allows for users to add new connectivity components, thus enabling the system to be further extended. The current version supports Bluetooth and Wi-Fi connectivity, but further connectivity support may be included, such as

IrDA or USB.

In order for the Communication module to choose which connectivity technology to use, it must take into consideration not only the different available technologies, but also each of those technologies' power consumption as well as the battery state of the mobile device.

Since this choice depends greatly on the limitations imposed by the state of the mobile device, it is this device's responsibility to decide which of the available technologies is best. The choice is based on the current power state of the mobile phone, and the performance/power consumption ratio.

It is the client's responsibility to perform the power consuming task of trying to establish a new connection to the server. The client will try to reach the server using every available technology. It uses broadcast to try to reach the server over Wi-Fi, and the device and service discovery services of the Bluetooth technology.

After setting up the initial connection, if the mobile device decides that a different technology must be used, the client disconnects and reconnects using the selected technology.

The newly established connection will be used by the Reconciliation module to execute the reconciliation process. The connection will only be active during the execution of this process, thus minimizing power consumption. The connection establishment process is repeated on request by the Reconciliation module. Previous discovery results are maintained in order to minimize the discovery overhead.

2.2.2 Storage Module

This section describes the architecture of the Storage module. This module is designed to provide an abstraction layer of the underlying file system for the FEW Phone FS, and to provide the means to detect and determine missed updates between system nodes.

In order for the Storage module to provide an abstraction layer for the underlying file system, it must be responsible for handling the intercepted file system calls made to the FEW Phone FS. For this purpose, the Storage module must export file system access methods, as represented in Figure 2.4. When the user accesses the file system, the methods provided by this module are executed.

For the Storage module to be able to handle these calls, it has to directly manage the objects stored in the FEW Phone FS. The objects maintained and managed by the Storage module are called *replicas*.

```
void mknod(String path);
void mkdir(String path);
void unlink(String path);
void rmdir(String path);
void rename(String from, String to);

Attr getattr(String path);
DirEnts[] getdir(String path);
long open(String path);
void read(String path, ByteBuffer buf, long offset);
void write(String path, ByteBuffer buf, long offset);
void flush(String path);
void release(String path);
```

Figure 2.4 Methods exported by the Storage Module.

Replicas

Each replica managed by this module, i.e. each file system entry, maintains the set of attributes presented in Figure 2.5. These attributes include, besides the basic attributes for

```
int uid;
String name;
long cTime;
long aTime;
long mTime;
long size;
VersionVector version;
boolean updatedFlag;
boolean isDirectory;
byte[] contentsDigest;
boolean isOriginalFlag;
ReplicaSource sources;
```

Figure 2.5 Replica's attributes.

objects in a file system (creation, last access and last modification dates, size, and name), the global unique identifier of the replica, the type of replica entry (file or directory), the version identifier of the replica, a secure hash of the contents of the replica, a modified flag, the sources of the original data of the replica, and a flag for marking the contents of the replica as the original one. The replica's version identifier, contents hash, modified flag, and the source of data are used by the reconciliation module when preforming the synchronization process.

The Storage module is designed to internally address replicas by their globally unique identifiers (GUID), thus allowing the system to deal with name updates in a simpler and deterministic way, as explained later. This forces the Storage module to be responsible for maintaining

the file system structure, since the intercepted file system calls address objects by name, and GUIDs do not maintain any structural information.

For marking replicas as updated, the Storage module uses the replica's version identifier. This is based on a version vector [20], and it is used for keeping a summary of the updates performed on the file. A version vector V for a file f is a set of pairs (Si,Ci). Each pair (Si,Ci) represents the number Ci of updates performed on f, at site Si. We define $V(Si) = Ci$ when $(Si,Ci) \in V$, and $V(Si) = 0$ when $(Si,\_) \notin V$.

The system can detect data updates, during the reconciliation process, by verifying that version vectors are different (i.e. $V_1 \neq V_2$ iff $\exists\ (S,C_1) \in V_1 : C_1 \neq V_2(S) \vee \exists\ (S,C_2) \in V_2 : C_2 \neq V_1(S)$). More specifically, two synchronized replicas, 'A' and 'B', of the same file have $V_A = V_B$ (i.e. $V_1 = V_2$ iff $\forall\ S \in V_1$, $V_1(S) = V_2(S) \wedge \forall\ S \in V_2$, $V_2(S) = V_1(S)$). An update to replica 'A' will leave $V_A > V_B$ (i.e. $V_1 \geq V_2$ iff $\forall\ (S,C_2) \in V_2\ \exists\ (S,C_1) \in V_1 : C_1 \geq C_2$). A concurrent update is detected when $V_A \not\geq V_B$ and $V_B \not\geq V_A$.

Unlike the common approach, the FEW Phone FS also uses an 'updated' flag to mark replicas as updated. This is used to prevent adding a new entry to the version vector by using existing entries to record the update. For example, consider two synchronized replicas of 'x', one in node 'A' and the other in node 'B'. The version vector has no entry for node 'A' and the entry for node 'B' is set to '1'. An update to the replica in node 'A' will mark that replica as updated, instead of creating a new entry on the vector (since the vector has no entry for that site). During reconciliation, the updated bit will signal the update, and, since the version of the object is the same in both nodes, the vector is incremented in the entry of 'B' instead of creating a new entry for 'A'. This allows the system to minimize the number of entries in the version vector, thus contributing for reducing space overhead, and improve scalability.

The impact of this improvement depends on the synchronization pattern, but if there is a large number of replicas that always synchronize with a small number of sites, only these sites need to keep an entry in the version vector. In our scenario, if the synchronization process always includes a single mobile phone, it is possible to keep only a single entry in the version vector, independently of the number of replicas.

The contents' secure hash is used to enhance the reconciliation process. This allows the system to detect identical replica contents, thus avoiding transferring them during the synchronization process, only synchronizing the metadata.

Metadata is synchronized by clearing the updated flag, and setting the version vectors of both replicas to the same state, as presented in Algorithm 2.1. This is done by comparing the

counter of each entry on both version vectors and setting it, in both version vectors, to the maximum value of these counters. This way, both version vectors will evolve to the same state.

Algorithm 2.1 Version vectors synchronization algorithm.

$VVlocal := localReplica.version$
$VVremote := remoteReplica.version$
$VVsync := \emptyset$
foreach $(Si, Ci) \in VVlocal$ **do**
    $otherCi := VVremote(Si)$
    $maxCi := \max(Ci, otherCi)$
    $VVsync := VVsync \cup \{(Si, maxCi)\}$
end for
foreach $(Si, Ci) \in VVremote$ **do**
    $otherCi := VVlocal(Si)$
    $maxCi := \max(Ci, otherCi)$
    $VVsync := VVsync \cup \{(Si, maxCi)\}$
end for

The 'sources' and the 'isOriginal' flag entries of the metadata are used when a file has been updated and its new contents have been obtained from a remote site. This is detected using the DSV module. If the contents have been obtained from a remote site, then the 'isOriginal' flag is set and the DSV module is used to retrieve that site's reference (URL). That reference is then added to the 'sources' entry of the replica's attributes, removing previous references since they are no longer valid. The use of this information will be explained is Section 2.2.3.

Since the Storage module is responsible for executing file system calls made to the FEW Phone FS, it must deal with data updates. The version identifier and update flag attributes, previously described, allow the system to deal with these updates, thus providing the means for the system to reach a globally consistent state. Possible updates include data updates and directory updates, as described next.

Data Updates

When a data update operation is executed, the Storage manager is responsible for updating the target replica's attributes. This is done by updating the last access and modified times, size, contents secure hash, sources and 'isOriginal' flag, if the new contents have been obtained from a remote site, and for marking the target replica as updated.

To mark a replica as updated the replica's version and/or 'updated' flag are used, as presented in Algorithm 2.2. This allows the reconciliation process to detect data updates by comparing these attributes, as explained in Section 2.2.3.

Algorithm 2.2 Marking updated replicas algorithm.

```
MarkReplicaUpdated(replica)
{
VVlocal := replica.version
Si := getLocalSiteID()
Ci := VVlocal(Si)
if Ci ≠ 0 then
    VVlocal(Si) := max({VVlocal(S) : ∀ S}) +1
end if
replica.updatedFlag := true
}
```

In order to optimize the reconciliation process, when a file is updated, all directories in the file's path are marked as updated, all the way to the file system's root, as presented in Algorithm 2.3. This simplifies reconciliation, since, if neither of two replicas of the same directory are marked as updated, then all entries in those replicas are synchronized, thus allowing the synchronization process to skip that directory. The updated flag is also used to avoid the propagation of updated information to already updated directories, thus allowing this process to stop when it detects a directory already marked as updated.

Algorithm 2.3 Parent update propagation algorithm.

```
MarkParentsUpdated(replica)
{
parent := getReplicaParent(replica)
if not(parent.updatedFlag) then
    MarkReplicaUpdated(parent)
    if not(isRoot(parent)) then
        MarkParentsUpdated(parent)
    end if
end if
}
```

Directory Updates

In order for the system to deal with directory updates, a logging structure is used. Each node's

log maintains the information of all directory updates executed in all nodes, thus allowing the log to be used by the reconciliation process. Each time a directory update is performed, a new log entry is added to this log. Possible directory updates are: create, delete, and rename file/directory. The Storage module is responsible for maintaining and managing the directory update log.

Each log entry has a unique identifier, and the description of the operation that generated the entry (including the operation identifier, the identifier, name and version of the updated replica, and the identifier of the updated replica's parent). This information allows the system to reproduce the operations in other nodes.

An entry's unique identifier is a pair (C_i, S_i), where C_i is a monotonically increasing counter and S_i is the node's unique identifier. This provides the means for totally ordering log entries (i.e. $(C_1, S_1) < (C_2, S_2)$ iff $C_1 < C_2 \vee (C_1 = C_2 \wedge S_1 < S_2)$), allowing the system to, in case of conflict, reach a consistent state during the synchronization process.

The operation identifier provides the system with information of which directory update operation was executed. The replica's identifier allows the system to unequivocally identify which replica was updated, and the replica's version is used to detect possible conflicts. The resolution of these conflicts is described is Section 2.2.3. Also, the identifier of the parent directory of the replica is used to solve possible naming conflicts. Consider the following example, two nodes, 'A' and 'B', have two replicas of the same directory, thus with the same GUID, 'X'. Node 'A' renames directory 'X' to 'Y', and creates a new directory 'X'. Node 'B' creates a new file 'Q' under the directory 'X'. In order to achieve the same state, the parent directory's GUID must be used by node 'A' to create file 'Q' under the directory 'Y' and not under the new directory 'X'. This way both nodes will reach the same state after the reconciliation process.

The log structure also maintains an associated version identifier. This version identifier is based on a version vector [20] that stores a summary of the updates produced by each node to the directory structure. Each entry of the vector represents the counter of the latest update made by that site.

2.2.3 Reconciliation Module

This section presents the design of the Reconciliation module and describes in detail the synchronization algorithm, as well as the conflict resolution protocol used by the FEW Phone FS.

Since file system updates can be divided into two distinct groups (directory updates and data

updates), the reconciliation algorithm is divided into two distinct phases: directory reconciliation and data reconciliation.

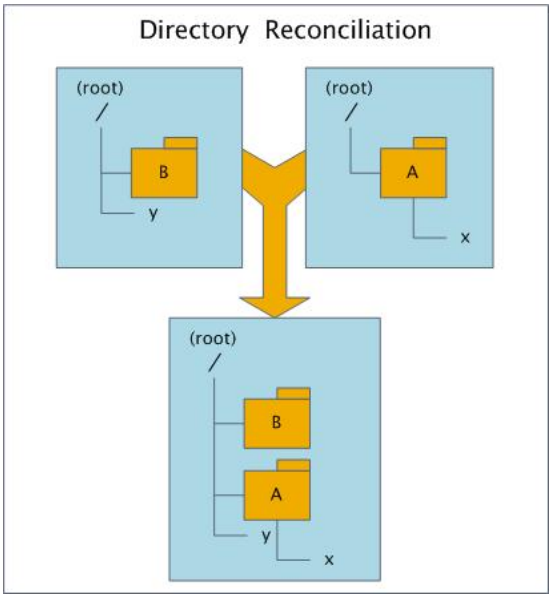

Figure 2.6 An example of directory reconciliation.

Directory Reconciliation

The purpose of directory reconciliation is to create a coherent directory structure from two, possibly conflicting, structures. For example, Figure 2.6 represents the result of the reconciliation of two replicas that concurrently evolved from an empty root directory. The result includes all files and directories from both replicas.

Algorithm 2.4 presents the directory reconciliation algorithm. The algorithm begins with each node exchanging the version of the directory update log. With this information, each nodes can detect the other node's unknown directory updates. The unknown updates are then exchanged in order to be executed.

The system will then execute these operations, respecting causal order in each replica. Since different replicas may execute updates in different orders, the system guarantees the convergence of both replicas by guaranteeing that every pair of concurrent operations commute. To this end the following approach is used for handling conflicting updates.

Algorithm 2.4 Directory reconciliation algorithm.

```
VVlocal := getLocalStorageLogVersion()
send(VVlocal)
VVremote := receive()
if VVlocal ≠ VVremote then
    remoteUnknownUpdates := getUnknownUpdates(VVremote)
    send(remoteUnknownUpdates)
    localUnknownUpdates := receive()
    foreach update in localUnknownUpdates do
        execute(update)
    end for
end if
```

Since the FEW Phone FS is designed to address replicas by their GUIDs, the create/create conflict does not exist, since different files have different GUIDs, independently of their names. Unlike the common approach, we allow for different objects to have the same name. As explained next, this simplifies conflict resolution and poses minimal overhead.

When more than one entry, in some directory, has the same name, the FEW Phone FS deterministically adds additional information to one of the entries, in order to distinguish them from one another. For example, when a node lists the contents of a directory, for instance directory '/A', and two files have the same name, '/A/b', the FEW Phone FS appends the GUID to the name of the entry with the highest GUID. This way, the system will show all entries with different names and, for two synchronized directories, the same file will have the same name. This way, when the user observes such a name, he is free to keep the system as it is, but he is expected to solve the conflict by renaming one of the files. The Storage module maintains the mapping between the modified name and the replica's GUID.

Delete/delete conflicts also do not pose any problem since the removal of an already removed replica will leave the system unchanged.

Rename/rename conflicts are resolved using the log entry's unique identifier. The rename operation with the highest identifier overrides the other. This way each node will reach the same state, thus leaving the system in a coherent state.

Rename/delete conflicts also do not pose any problems since replicas are addressed by ID, this way the delete operation will delete the replica leaving the system in a consistent state.

The most complex case involves the delete operation and the implicit operation of concurrently updating another replica of the file. The execution of a delete operation only causes the

deletion of the file if the local replica's version is lower or the same than the one on which the delete operation was executed (to check this, the delete operation records the version of the replica at the time of delete). Otherwise, the operation has no effect.

This leaves the log structure in both nodes synchronized, but does not guarantee the consistency of the file system structure, because a file may have been previously deleted in one of the replicas. The data reconciliation algorithm is responsible for dealing with this possible problem, as described next.

Data Reconciliation

The purpose of data reconciliation is to synchronize the contents of all replicas in the file system, thus leaving each replica's contents identical in both nodes.

Algorithm 2.5 Data reconciliation algorithm.

```
Reconcile(dir)
{
localEntriesInformation := getEntriesInformation(dir)
send(localEntriesInformation)
remoteEntriesInformation := receive()
if localEntriesInformation ≠ remoteEntriesInformation then
    resolve(remoteEntriesInformation)
end if
foreach replica in dir do
    send(replica.version)
    rmtRepVersion := receive()
    send(replica.updatedFlag)
    rmtRepUpdtFlg := receive()
    if DetectReplicaUpdate(replica, rmtRepVersion, rmtRepUpdtFlg) then
        if replica.isDirectory then
            Reconcile(replica)
        else
            SelectReplicaUpdate(replica, rmtRepVersion, rmtRepUpdtFlg)
        end if
    end if
end for
}
```

The data reconciliation algorithm, as presented in Algorithm 2.5, is designed as an iterative algorithm, starting at the file system's root. Since file updates are signaled all the way to the root

of the file system, the versions of the root, as with any other directory, are used to determine the existence of updates in the subtree rooted in that directory.

Since different replica versions of a directory are the result of updates to one or more entries of that directory, this allows the algorithm to skip entire portions of the file system, during the data reconciliation process, every time directory versions are identical.

Each iteration begins with both nodes exchanging the information of all entries of the directory. This information includes the GUID, name, and type (file or directory) of the entry. This way, possible update/delete conflicts are resolved by re-creating the respective deleted objects. The creation of these objects does not affect the Storage log.

Whenever an update to a file replica is detected, the reconciliation algorithm will propagate the update. Data updates are detected by comparing version vectors and update flags, for two replicas of the same file, as presented in Algorithm 2.6.

Algorithm 2.6 Replica update detection algorithm.

DetectReplicaUpdate(*localReplica*, *rmtRepVersion*, *rmtRepUpdtFlg*)
{
return $localReplica.version \neq rmtRepVersion \vee localReplica.updatedFlag \vee rmtRepUpdtFlg$
}

To determine the latest/valid version of a file, the reconciliation process uses a deterministic method, as presented in Algorithm 2.7. First, version vectors are updated, taking into account each replica's update flag (for simplicity, we omit our optimization of using the remote entry of the version vector for recording updates). After this, version vectors are used to determine the latest version. If there are no concurrent updates, the most recent version will dominate the other, and the update is propagated. For two replicas, 'A' and 'B, when $V_A > V_B$ the state of replica 'A' is propagated to replica 'B'. When neither version vectors dominate, then the updates are found to be concurrent, thus conflicting.

Algorithm 2.8 is used to resolve conflicting updates. The version to keep is the one that reflects the update that has the largest identifier using the Lamport [16] order on pair (counter, site ID). This approach guarantees that the same version wins, independently of the number of replicas involved. The "loser" version will be moved to a special "lost and found" directory, allowing users to restore it if needed, and the "winner" version will replace it.

The data reconciliation process also takes advantage of the DTC module. When sending multimedia data to the mobile phone, the DTC is used to re-encode the data before transferring

Algorithm 2.7 Update selection algorithm.

```
SelectReplicaUpdate(localReplica, rmtRepVersion, rmtRepUpdtFlg)
{
localSiteID := getLocalSiteID()
remoteSiteID := getRemoteSiteID()
if localReplica.updatedFlag then
    localC := max({localReplica.version(S) : ∀ S })
    localReplica.version(localSiteID) := localC + 1
end if
if rmtRepUpdtFlg then
    remoteC := max({rmtRepVersion(S) : ∀ S})
    rmtRepVersion(remoteSiteID) := remoteC + 1
end if
if localReplica.version > rmtRepVersion then
    send(localReplica.contents)
else if localReplica.version < rmtRepVersion then
    localReplica.contents := receive()
else
    ResolveUpdateConflict(localReplica, localReplica.version, rmtRepVersion)
end if
}
```

Algorithm 2.8 Conflict resolution algorithm.

```
ResolveUpdateConflict(localReplica, localRepVersion, rmtRepVersion)
{
(LC, LS) := localRepVersion[0]
foreach (C_i, S_i) ∈ localRepVersion do
    if (C_i, S_i) > (LC, LS) then
        (LC, LS) := (C_i, S_i)
    end if
end for
(RC, RS) := rmtRepVersion[0]
foreach (C_i, S_i) ∈ rmtRepVersion do
    if (C_i, S_i) > (RC, RS) then
        (RC, RS) := (C_i, S_i)
    end if
end for
if (LC, LS) > (RC, RS) then
    send(localReplica.contents)
else
    moveToLostNFound(entry)
    localReplica.contents := receive()
end if
}
```

it. This way, the smaller transcoded version is transferred from the computer to the mobile phone, and the reference to the original replica is added to the server's replica's sources attribute. Note that, as these files are usually not changed, this process will only occur during the first synchronization.

When receiving any contents from the mobile phone, if the replica contains any source references, these are used to try to transfer the full-fidelity contents of the replica. Recall that these references may also keep the sources from remote sites added by the DSV module. If this process is successful, a new reference is then added to the server's replica's source's attribute, thus improving availability. If not, the client must transfer the transcoded version from the server, and use the 'isOriginal' flag to mark the newly created replica as "not original". The system will periodically try to retrieve the original contents from one of the available sources, until it is successful.

The same approach is used for all file objects kept in the FEW Phone FS. Every time a replica is created, a new reference is added to the source attribute of the server's replica. This way the synchronization will depend less on the mobile phone, since the contents of the replicas can be transferred directly between nodes, in a peer-to-peer interaction, thus minimizing power consumption and improving the performance of the system. The number of references per replica can be used to determine the need for maintaining the contents of the replica on the mobile phone, thus improving the storage capacity of the system. In the limit, the server can be used for storing only the URLs of the available replicas, thus increasing the total storage capacity of the system.

Since the FEW Phone FS uses primarily the client-server model, this allows the reconciliation process to be executed every time the server is detected by a node. This means that, whenever the system detects the presence of the user (mobile phone), it will automatically start the reconciliation process. This process can also be explicitly started, thus allowing a user to execute the process before disconnecting. Since the server is a mobile device, this presents as a reasonable method, although it forces the user to explicitly start the process. In order to solve this, the reconciliation process can be made more frequent.

2.2.4 Data Transcoding Module

This section describes the architecture of the Data Transcoding module. This module allows for multimedia contents to be re-encoded in order to better adapt to a mobile device's specifications.

To this end, this module is designed to transcode multimedia contents according to a mobile device's specification. For example, a 2 mega pixels digital photograph with 32 bits color depth, which can only be presented with a resolution of 320x240 and a color depth of 16 bit in most high-end mobile phones, can be transcoded to better "fit" the mobile device's specifications.

To this end, the system includes for each data type an associated program and configuration to perform the data transcoding. This way the user can define specific rules for different data types. For example, a user may decide to keep digital photos with a higher resolution than the mobile phone's specifications, thus allowing him to use the mobile phone for displaying these photos with a higher level of detail.

In the case of a digital image, a resize and a reduction in the color depth of the image is sufficient. In case of digital audio files, a re-encode operation is done to reduce the sound quality of the file. In case of a digital movie file, resize and re-encoding operations will be performed in order to adapt the file's contents to the defined configuration. This allows for the data volume of multimedia contents to be reduced, which will lead directly to less storage consumption.

In order to choose which files need to be re-encoded, the original multimedia files need to be analysed, i.e. comparing the quality of these files to the specifications on the mobile device.

2.2.5 Data Source Verification Module

This section describes the architecture of the Data Source Verification module. This module allows for data contents to be verified in order to detect the origin of that contents.

The DSV module is designed to check the source of a file's contents. If that contents have been transferred from a remote site (e.g. WWW), this module is used to obtain that site's URL.

To this end, the DSV module intercepts network communications, storing the URLs of the accessed contents. The URLs kept by the DSV module are associated with the secure hash of their contents. The Storage module queries this module to find if the a file's contents have been obtained from a remote site, and to retrieve that site's URL.

3. Implementation

The previous chapter presented the requirements of the FEW Phone File System as well as the design and architectural options made to address those requirements. This chapter discusses the implementation of the system prototype, as well as a discussion on the relevant problems found during this process.

3.1 Environment

The FEW Phone File System prototype was completely written in the Java programing language. The reason for using the Java platform in the mobile device was that most mobile devices support it, allowing the system to target a wide variety of devices.

The requirements imposed by the system's design, related to the usage of multiple connectivity technologies, also favored this choice since the Java language features many packages for addressing the multiple connectivity needs of the FEW Phone FS. The Standard Java environment (J2SE) features packages that deal with network connectivity, and can be extended through the use of different modules for specific technologies such as Bluetooth, IrDA, or even USB.

In the desktop, we have decided to use the Linux system because it includes the FUSE [11] system, that provides a simple solution for the interception of file system calls, which was a requirement for our system. For using FUSE with the Java language, the FUSE-J [17] wrapper was used.

This solution should work without modification in other systems supporting FUSE (e.g. MacOS X) and with few changes in systems with similar file system interception methods (e.g. Dokan [1] for MS Windows).

3.2 Storage Module

As described in Section 2.2.2, the Storage module is responsible for interacting directly with a node's file system in order to maintain and manage the node's replicas.

The FEW Phone File System uses the Java wrapper for the FUSE module, named FUSE-J.

This module allows the creation of a new file system that is managed by a user-level process. Thus, when mounting the new file system on a given directory, any file operation on that directory is intercepted and redirected to our system.

Since the FEW Phone FS is designed, mainly, for a personal environment, it can be mounted initially into a directory already containing a user's personal files. In this case, the Storage module starts by moving the contents of the directory into the FEW Phone FS storage directory, that is a hidden directory kept on the user's home directory. This operation will automatically create the metadata for each of the files stored in the user's directory, as if they were newly created replicas. Since the creation of replicas automatically updates the log kept by this module, this operation will also add new entries to the log, thus allowing for the reconciliation process to run immediately.

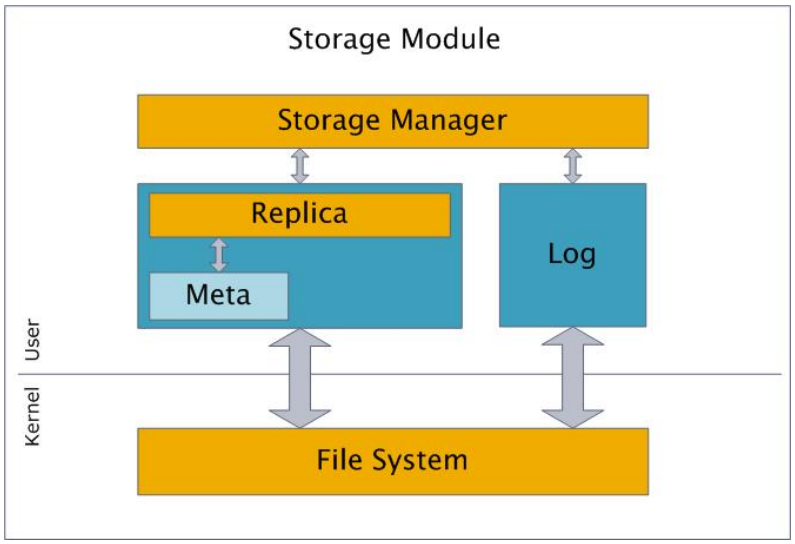

Figure 3.1 FEW Phone File System's Storage module architecture.

Figure 3.1 presents the components of the Storage module. The following sections describe these components and their interactions. Section 3.2.1 describes the lowest level component of the this module, the Replica component. Section 3.2.2 presents the logging component of the Storage module, and Section 3.2.3 describes the Storage Manager.

3.2.1 Replica

The Storage module is responsible for managing data by interacting directly with the storage system. The objects managed by this module are called Replicas. This section describes some of the implementation details of the Replica component. Common files or directories are generally addressed as file system objects.

For each object, the FEW Phone FS keeps two distinct files in the file system, one file maintains the replica's data and the other maintains the replica's metadata. This is hidden from other modules by building a wrapper that aggregates these two files into a single abstraction.

The system must guarantee the uniqueness of an GUID. For this purpose, GUIDs are generated as a secure hashcode of the concatenation of the replica's creation path name, creation date, and the node's unique identifier.

File replicas are stored, in persistent memory, in a hidden directory on the user's home directory. For each object (file or directory), files are stored using the GUID of the objects. All files are stored in the same file system directory.

Directory replicas differ from file replicas in their data contents. A directory contains the GUIDs of the objects contained in that directory.

Version vectors are implemented as maps. Each entry of the map is the association of a node's unique identifier and the update counter for that node. The current version of the FEW Phone FS uses a node's Bluetooth or network address as the node's unique identifier.

The 'sources' field is used to maintain references to other existing replicas of the specific object. This is kept as a list of the URLs for each of the other replicas. This information is also used for optimization purposes. The current version of the system adds a new URL to the server's replica metadata every time a new replica of the object is created, as described in Section 2.2.3.

The replica's metadata is automatically updated whenever the replica is updated, in this case the Metadata's modified time, contents digest, and version vector and/or updated flag of the replica. Also, the URLs for other replicas of that file are discarded since they are no longer valid.

In order to reduce overhead, a replica's metadata is only updated and written to persistent storage when the replica starts and stops to be modified.

3.2.2 Operation Logging

The Log component is used to store directory updates preformed to the system. These operations must be stored, in order for the system to reproduce them in other nodes during the reconciliation process.

To achieve this, the Log component maintains its version and a list of *Log Entries*. The log's version is kept in a version vector using a map, as explained before.

Each Log Entry contains the needed information for reproducing the operation in other nodes. A Log Entry has an identifier that is composed of a monotonically crescent counter and the node's unique identifier. This allows for totally ordering the log structure in order to achieve consistency, as explained before.

The log structure is kept in persistent storage, and it is updated every time a directory update is added. In the current version, all data is written to persistent storage. A more efficient version could be implemented in the future.

3.2.3 Storage Manager

The Storage Manager component's primary function is to handle the file system calls redirected by the interception layer. Thus, this component is responsible for managing the FEW Phone File System structure. In order for the intercepted file system calls to be handled, this component implements the FileSystem interface defined by the FUSE-J module. This interface presents the standard methods for file system access, as presented in Figure 3.2.

FUSE's redirected file system calls are based on path names. As internally object replicas are named by their GUIDs, the Storage Manager is responsible for mapping path names and GUIDs. This information can be obtained by solving names using the directory objects, as usual. Additionally, the system keeps a cache of this information in memory for improving performance.

The Storage manager is responsible for adding a new entry to the operations Log, when a directory update operation is made to the file system. It is also responsible for marking file replicas, and all its ancestors, as updated, when a file update is made.

```
// File System information method
FuseStatfs statfs();

// File System directory access methods
void mknod(String path, int mode, int rdev);
void mkdir(String path, int mode);
void unlink(String path);
void rmdir(String path);
void rename(String from, String to);
FuseDirEnt[] getdir(String path);

// File System's file access methods
long open(String path, int flags);
void read(String path, long fh, ByteBuffer buf, long offset);
void write(String path, long fh, boolean isWritepage, ByteBuffer buf, long offset);
void flush(String path, long fh);
void release(String path, long fh, int flags);

// File System's objects attibutes
FuseStat getattr(String path);
```

Figure 3.2 The FileSystem Interface.

3.3 Communication Module

The previous section discussed the implementation details of the Storage module. This section will present the implementation details of the Communication module.

The main purpose of this module is to create the communication channel, between the client and the server, to be used during the reconciliation process.

This module's implementation has some differences from its original design. This was forced by the limitations imposed by the mobile phone's platform. The following section describes these limitations, and the solutions found to solve them. Section 3.3.2 describes the implementation details of the module.

3.3.1 Implementation Problems & Solutions

During the implementation process, the behaviour of the mobile phone imposed a different solution to the Communication module. The main problem found was related to the behaviour of the server's socket creation. In the original design, the mobile phone was responsible for choosing and creating the server endpoints of the communication channels. This meant the creation of a server socket in case of the Wi-Fi technology, and the creation of a new service in case of the Bluetooth technology.

However, when creating a new server socket on the mobile phone, the device's operating system assigned it a local address that could not be reached from the network. In order for the server socket to be accessible it was necessary to first establish a connection to some remote site.

To address this problem, the implemented solution combines the server side of the Bluetooth connectivity (service creation) with the client side of the Wi-Fi (socket creation), while the client's design combines the client side of the Bluetooth connectivity (device and service discovery) and the server side of the Wi-Fi (server socket).

At startup, the mobile phone creates the Bluetooth service. The client must search for the mobile phone, and must create a new server socket (when LAN connectivity is available).

Once the server has been discovered, a new Bluetooth connection is established. This connection is used for the client to notify the mobile phone of which technologies are available.

The mobile phone is still responsible for determining which connectivity technology is best for establishing the final connection. After the decision is reached, the server notifies the client on which technology to use. If the choice has favored the Wi-Fi technology, the mobile phone will establish a new communication channel to the socket created by the client. If the Bluetooth technology is chosen, the already established communication channel is preserved.

The final communication channel will then be used during the reconciliation process. As described in Section 2.2.1, in order to reduce power consumption, the communication channel is only active during the execution of the reconciliation process. A new communication channel is created, on demand by the Reconciliation module, whenever the reconciliation process is executed. The previous discovery results are maintained in order to minimize the search overhead.

3.3.2 Implementation

The implementation of the Communication module is based on the description presented on the previous section. The Communication Module is composed of two different components, as presented in Figure 3.3. The Connection Manager is responsible for creating the communication channel between the client and the server, while the Communication Manager is responsible for dealing with the different data types transferred between nodes during the reconciliation process, thus providing an abstraction layer for the Reconciliation module.

In order to detect devices and services, the client's Connection Manager uses the device

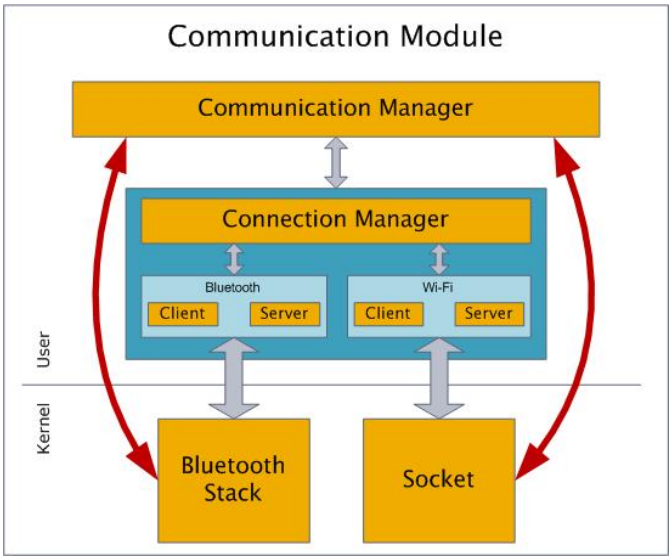

Figure 3.3 Generic architecture of the Communication module.

and service discovery service offered by the Bluetooth technology. For this purpose, the FEW Phone FS's client application uses the Bluecove [31] implementation of the JSR 82 API, while the server relies on the J2ME implementation of this API.

The mobile phone is responsible for creating the new service, and for maintaining a Thread listening for incoming connections. The client node is responsible for running the discovery algorithm. Since the client side does not have energy restrictions, the algorithm is run consecutively until a device with the specific service is discovered.

3.4 Data Transcoding Module

This section describes the Data Transcoding Module. As presented before, this module is responsible for re-encoding multimedia contents. The quality of the re-encoding is based on the configuration defined in the mobile device, typically suitable for its specifications.

In order for the Data Transcoding module to re-encode multimedia files to the mobile device's specification, the current version of the module relies on user defined specifications. The user is responsible for defining the encoding quality for each type of multimedia files. This gives users the possibility of defining the encoding quality based on their needs.

When a client's compatible file is detected during the reconciliation process, and before it is transferred to the mobile phone, this module is called upon and a new version of the file is created and sent to the mobile phone.

The URL for the original file is added to the mobile phone's replica's metadata. This allows for later reconciliation processes to request the original contents from a remote node, instead of relying on the adapted version stored on the mobile phone.

Before transferring a transcoded replica from the mobile phone to a client node, the URLs kept in the replica's metadata are used to request the original contents from other client nodes. If it is not possible to obtain the original contents, the transcoded version is stored in the local node as well as the references of the original data, and the 'isOriginal' flag is used to signal that the replica's contents is not the full-fidelity version. Before an open request is made to an adapted replica, the system will try to retrieve the original contents from one of the available sources. This process is repeated until the original contents have been obtained. When this happens the replica is marked as original, by setting the 'isOriginal' flag to true.

Whenever a new full-fidelity replica of a multimedia file is created in a node, which means that the server has received a request for a multimedia file and that the original content was obtained successfully, the newly created replica's URL is added to the mobile phone's replica metadata. This will increase the availability of the multimedia resources stored on the mobile phone.

The current version of this module only allows for '.jpg' and '.png' image files to be re-encoded, by using the native Java support. In the future, it is expected that this module can be used to also re-encode movie and audio files, as well as further support for image files, by using external tools.

3.5 Data Source Verification Module

This section presents the implementation of the Data Source Verification module. The main purpose of this module is to determine the origin of a given data contents. This is used by the FEW Phone FS to detect if a given replica's contents has been obtained from a remote site.

The current version of the module is based on a small HTTP proxy application that its only function is to maintain a map of contents digests associated with URLs.

This module is called whenever a replica has been updated. This way, if the replica's contents have been obtained from a remote site, that site's URL will be added to the replica's metadata. This URL will then be propagated to the server during the synchronization.

The current version of this module only allows for HTTP contents, obtained with GET operations, to be verified. In the future, it is expected that this module can be used with additional protocols.

4. Evaluation

This chapter presents the evaluation performed with the prototype of the system. Due to the characteristics of the FEW Phone File System, the tests made are mostly related to access times and reconciliation times measurements.

In order to better evaluate the system's performance, both the server (mobile phone) code and the client (personal computer) code were executed in workstations. This allowed us to gather reference results for both Bluetooth and Wi-Fi technologies during the synchronization process. These results were then compared to the ones obtained running the server code in mobile devices.

An Apple iPod Touch 1G was used to obtain results using Wi-Fi technology for the synchronization process. To this end, the server code was executed in this device. This allowed us to obtain results running the system in other mobile devices that, although similar to mobile phones (in fact, the Apple iPod Touch is a downgrade version of the Apple iPhone, where the telephony functionalities are not present), present some different characteristics. In fact, battery duration for the iPhone is higher than the iPod's.

Unfortunately it was impossible to perform a complete performance evaluation of the prototype with J2ME phones due to the mobile phone's behaviour. Java developed applications for mobile phone's (J2ME applications) require a digital signature in order to fully access the underlying hardware platform. Unsigned J2ME applications require explicit user authorization, for example, every time a file system access is attempted. These requests made it impossible to obtain performance results in these devices, since it was not possible to acquire a license for digitally signing the developed mobile application on time. Thus the presented results for Bluetooth and Wi-Fi synchronization were obtained using only the already mentioned methods.

The results presented in the following sections are always the average of 8 runs, after removing the highest and lowest obtained results during those runs. All read and write tests were preformed with a clean file system cache (after the system was rebooted).

Section 4.1 presents results related to power consumption in the mobile device, while using wireless technologies. These tests were performed to serve as input for the decision on which technology to use.

Section 4.2 presents performance of single file read and write operations performed in the FEW Phone File System, and Section 4.3 presents the results for multiple files read and write operations. Finally, Section 4.4 presents performance results for the reconciliation process.

The presented results for the client application were obtained in an environment with the specifications presented in Table 4.1. During testing, the FEW Phone File System cache was stored in the same hard drive and file system as the local file system.

Operating System:	Linux Ubuntu 8.04.2
Kernel version:	2.6.24-23
CPU:	Intel® Core™2 Duo T7200 @ 2.00 GHz
RAM:	2.00 GB
File System:	ext3
FUSE version:	2.7.2
FUSE-j version:	2.4 pre-release 1
Bluetooth:	v2.0 + EDR
Wi-Fi:	802.11g

Table 4.1 Evaluation environment specifications.

4.1 Power Consumption

This section presents the results for power consumption tests. This evaluation has the intent of determining each mobile device's energy consumption, and associated battery duration, for intensive communication tasks.

This test was performed using a Nokia N82 mobile phone, an Apple iPod Touch 1G and a PC with the presented specifications. The Nokia N82 mobile phone supports Bluetooth 2.0 + EDR connectivity, while both devices support the 802.11g protocol for Wi-Fi connectivity.

The performed test was designed to stress communications between the PC and the mobile devices, while measuring battery duration. Two scenarios were studied for transferring data between the mobile devices and the computer. The first, with an application transferring fixed size data packets between the two nodes, and the second using packets of different sizes. The communication pattern for both applications is identical. The client (computer) is responsible for initiating communications by sending a message to the server (mobile device). The server is responsible for sending it back to the client only after it as been fully received, in a send and wait scheme. For the second scenario, the size of the message is randomly chosen by the client, with an equal probability. The communication channel established between the two nodes relied on one of two available wireless technologies, Bluetooth or Wi-Fi, the latter in infrastructure

mode.

To measure each mobile devices' battery life, tests were started with the batteries fully charged and measured the time it took for each mobile device to switch off due to low battery.

Table 4.2 presents the results when using packets with fixed packet size - 4 kBytes in this case. The obtained results include the number of exchanged packets between the mobile device and the workstation, total volume of data transferred during communications, the average transfer speed and the total battery life of the mobile device.

	Device		
	Nokia N82		**iPod Touch**
	Bluetooth	Wi-Fi	Wi-Fi
Transferred Packets	78 733	147 920	102 012
Transferred Bytes	$\approx 314\ MBytes$	$\approx 592\ MBytes$	$\approx 408\ MBytes$
Average Speed	$\approx 15\ kB/s$	$\approx 53\ kB/s$	$\approx 61\ kB/s$
Battery Duration	$\approx 5h48m$	$\approx 3h12m$	$\approx 1h51m$

Table 4.2 Battery life results for fixed sized packets.

From these results it is possible to observe that the power consumption for Wi-Fi communications is much higher than for Bluetooth communications. In the Nokia mobile phone, the battery duration when using Wi-Fi communications lasted approximately 50% less than when using Bluetooth, with the latter technology also transferring approximately half the data volume of the former. These results were expected, since Wi-Fi's bandwidth is much higher than Bluetooth's. Both technologies' power consumption is considerable. As presented in Table 4.3 the battery duration for each device when idle are much higher than the ones obtained using communications. These results were obtained with connectivity options switched off for both devices.

In the second scenario, small size packet have 100 Bytes, medium size packet have 1 kBytes, and large size packet have 10 kBytes. The collected results are presented in Table 4.4.

Comparing these results to the ones obtained using fixed-size packets, it is possible to observe a degradation of performance when using Wi-Fi technology. This is pretty noticeable when comparing the results obtained using the Nokia mobile phone. On the other hand, the

Device	
Nokia N82	**iPod Touch**
≈ 7 *days*	≈ 14 *hours*

Table 4.3 Battery life in idle.

		Device		
		Nokia N82		**iPod Touch**
		Bluetooth	Wi-Fi	Wi-Fi
Transferred Packets	100 *Bytes*	47 217	13 585	31 945
	1 *kBytes*	47 565	13 918	32 604
	10 *kBytes*	47 681	13 837	32 682
Transferred Data		≈ 529 *MBytes*	≈ 154 *MBytes*	≈ 363 *MBytes*
Average Speed		≈ 23.8 kB/s	≈ 12.6 kB/s	≈ 55.5 kB/s
Battery Duration		$\approx 6h12m$	$\approx 3h21m$	$\approx 1h49m$

Table 4.4 Battery life results for multi-sized packets.

results obtained using Bluetooth technology have improved. In order to better understand this we have tested the same application running both the client and the server in two PCs. The obtained results are presented in Table 4.5.

From the obtained result it is possible to see the degradation of performance for both technologies when the message size is reduced. This is pretty noticeable in the Wi-Fi results.

Taking into account these results it is possible to conclude that higher volumes of data result in better network usage. This allowed us to conclude that the Wi-Fi technology is possibly a better choice during the initial synchronization process, while during small synchronizations (where a large number of small messages are exchanged) the Bluetooth technology is sufficient, taking into account the difference in performance between the two technologies.

In the future, it would be possible to try to adapt the connection protocol for taking into consideration these results, for example by batching communications, or using both technologies simultaneously.

	Packet Size	Bluetooth	Wi-Fi
Average Speed	512 *Bytes*	10 kB/s	6 kB/s
	1 *kBytes*	20 kB/s	11.5 kB/s
	4 *kBytes*	45 kB/s	310 kB/s
	Multi Size	45 kB/s	65 kB/s

Table 4.5 Speed comparison for different size messages.

4.2 Single File Read and Write

This section presents the results obtained for sequential read and write operations performed using the FEW Phone File System. The results presented in Table 4.6 measure the execution time for copying files, with different sizes, to and from the FEW Phone File System root directory.

File size	Copying file from				
	Local FS to			FEW Phone FS to	
	Local FS	**FEW Phone FS**	**Overhead**	**Local FS**	**Overhead**
466 *Bytes*	0.002s	0.017s	8x	0.043s	≈22%
100 *kBytes*	0.031s	0.037s	≈19%	0.045s	≈45%
553 *kBytes*	0.073s	0.082s	≈12%	0.115s	≈57%
4.9 *MBytes*	0.366s	0.388s	≈6%	0.496s	≈36%
10.1 *MBytes*	0.760s	0.763s	≈0.3%	0.772s	≈1.5%
43.1 *MBytes*	2.404s	2.739s	≈14%	2.791s	≈16%

Table 4.6 Measured values for single file read and write operations.

From the obtained results it is possible to observe the overhead imposed by the system for writing operations. This is mostly because of the need to compute the contents' hash and to update the information for reconciliation. Despite the additional computation, the overhead is low (up to 15%), allowing it to go unnoticed for normal operations. Future work can help to reduce this overhead by adding some optimizations to the system.

Reading operations show, for small sized files, some overhead. For larger files, the FEW Phone File System presents a good performance, comparable to the local file system's. This

was expected since the overhead imposed by the system for read operations is low.

4.3 Multiple Files Read and Write

This section presents the results for compressing and decompressing packed archives into and from the FEW Phone File System, thus testing the overhead for accessing a directory tree with files of different sizes. The results presented in Table 4.7 were obtained using the 'tar -xzf' command on different sized packets.

Archive Size	**Number of Files**	**Unpacked Size**	Extracting files to		**Overhead**
			Local FS	**FEW Phone FS**	
100 *kBytes*	84 files	648 *kBytes*	0.082*s*	0.243*s*	$\approx 296\%$
495 *kBytes*	89 files	2.3 *MBytes*	0.110*s*	0.367*s*	$\approx 334\%$
151 *MBytes*	1108 files	171 *MBytes*	16.876*s*	25.590*s*	$\approx 151\%$

Table 4.7 Measured times for extracting files from an archive.

These results have shown a considerable overhead when compared to the values obtained from the extraction of the same contents to a local file system directory. The reason for this is the overhead of the systems maintenance of, not only the directory updates log, but also for the propagation of update information for reconciliation use from children to parents. This is one aspect that should be addressed in some future work.

Total Files Size	**Number of Files**	Adding files from		**Overhead**
		Local FS	**FEW Phone FS**	
648 *kBytes*	84 files	0.193*s*	0.235*s*	$\approx 22\%$
2.3 *MBytes*	89 files	0.363*s*	0.447*s*	$\approx 23\%$
171 *MBytes*	1108 files	14.644*s*	18.068*s*	$\approx 23\%$

Table 4.8 Measured times for adding files to an archive.

Table 4.8 presents the results for adding the extracted files into a new archive, using the 'tar -czf' command. The differences are much smaller than the ones verified earlier, since the imposed overhead for reading operations, as discussed in the previous section, is much smaller than for write operations.

These results seem to confirm that the overhead for the unpacking of the archive is mostly imposed by the management of the directory structure.

4.4 Reconciliation

This section presents the results for the reconciliation process. These tests were preformed on two different set sizes, with a clean server side cache (first reconciliation), and relying on both Bluetooth and Wi-Fi connectivity. The obtained results are presented in Table 4.9.

Number of Files	**Total Size**	**Reconciliation Time**		
		PC to PC		**PC to iPod**
		Bluetooth	Wi-Fi	Wi-Fi
84 files	648 *kBytes*	17.276*s*	6.876*s*	35.762*s*
89 files	2.3 *MBytes*	30.158*s*	8.856*s*	51.706*s*

Table 4.9 Measured first reconciliation times.

These result show, once again, the limitations of the Bluetooth technology when compared to Wi-Fi. It is also possible to observe that the difference in performance between these two technologies grows with the total volume of data. This had already been seen in previous results. From the obtained results it is also possible to conclude, since the total number of files is in the same order for both sets, and the total volume of data is more than twice the size, that the management of the directory structure, i.e. the replaying of the directory update log operations, seems to impose some non-negligible overhead.

The results for the reconciliation of two synchronized nodes are presented in Table 4.10. From these results it is possible to see that the overhead for this process is low. This is only possible since the FEW Phone FS uses the version of the log structure to detect missed updates during the directory reconciliation phase, and only compares the roots' versions during the data reconciliation phase.

In Table 4.11 are presented the results obtained during the synchronization of a second PC with the iPod Touch, after the initial synchronization. These results present a comparisson between relying solely on the server, for retreiving replica contents, with peer-to-peer interaction.

These results show the benefits of using the Data Source Verification module. The total volume of data, when using peer-to-peer interaction, was transfered directly between the two

Reconciliation Time		
PC to PC		**PC to iPod**
Bluetooth	Wi-Fi	Wi-Fi
0.479s	0.249s	0.396s

Table 4.10 Measured reconciliation times for synchronized nodes.

Number of Files	**Total Size**	**Reconciliation Time**		**Gain**
		with Peer-to-Peer	without Peer-to-Peer	
84 files	648 *kBytes*	14.877s	16.921s	$\approx 14\%$
89 files	2.3 *MBytes*	26.338s	30.393s	$\approx 15\%$

Table 4.11 Measured second reconciliation times.

PC, using only the mobile device for determining the available data sources. When compared to the results obtained using only the client-server interaction we can see a reduction of the reconciliation time of approximately fifteen percent.

Table 4.12 presents the results obtained during the synchronization of multimedia contents between a client PC and a clean side server running in the iPod Touch. This test was performed using two 15 mega pixels digital photos, with a resolution of 4752x3168. Each of these pictures had approximately 4 MBytes. The two photos were transcoded to a resolution of 800x533, with approximately 33kBytes.

Number of Files	**Total Size**	**Reconciliation Time**		**Gain**
		with Transcoding	without Transcoding	
2 files	8 *MBytes*	8.549s	27.761s	$\approx 325\%$

Table 4.12 Measured multimedia reconciliation times.

The obtained results show the benefits of using the Data Transcoding module. The presented results for the reconciliation without transcoding, were obtained from synchronizing the same two photos, but renamed to a different extension. This way the transcoder did not detect them as “transcodable”.

From the obtained evaluation results, it is possible to conclude that the overhead of the reconciliation process using either technology is significant. The performance of this process can be improved with future optimizations. The use of both the Data Transcoding and the Data Source Verification modules has proved to be beneficial for the performance of the system,

reducing not only the synchronization time, but also greatly reducing the communications and storage overhead.

5 . Related Work

In this chapter, we will present a general overview of some of the most significant systems that relate with the FEW Phone File System.

In the first section, some solutions related to mobile storage systems are described. The second section addresses synchronization and reconciliation solutions. The third section presents other work related with data management for mobile systems that might be relevant to our work.

5.1 Mobile Storage Systems

The following sections present a general overview of mobile storage systems that address some of the same problems addressed in our system.

5.1.1 Pangaea

Pangaea [25] is a wide-area file system that supports data sharing among a community of distributed users. The system is designed to build a unified file system across a federation of widely distributed computers connected by dedicated virtual private networks.

In order to cope with continuous reconfiguration (users moving, companies restructuring, network topology changes), the authors have defined three goals for Pangaea:

- Performance: the system was designed to provide file access performance identical to local file systems, thus hiding the wide-area network latency.
- Availability and autonomy: not depending on the availability of any node, Pangaea was designed to automatically adapt to server addition, removal, failure and network partitioning.
- Network economy: reducing the use of wide-area networks, Pangaea transfers data between nodes in physical proximity, thus reducing latency and saving network bandwidth.

Pangaea relies on pervasive replication, creating file and directory replicas whenever a user tries to access a file not present in a local node. This design overrides the "single master" design of most distributed file systems, since all nodes act as both clients and servers, thus allowing

for: replicas to be read or written at any time in any node; and updates to be exchanged between replicas in a peer-to-peer fashion.

Pervasive replication allows for high performance since it serves data from the server closest to the point of access, high availability by letting each server contain its working set, and network economy by transferring data among close-by replicas.

Pangaea's replica management satisfies three goals: support a large number of replicas in order to maximize availability; need to manage replicas of each file independently; and support dynamic addition and removal of replicas. For Pangaea to address these goals, it maintains a sparse but strongly connected and randomized graph of replicas for each file. These graphs are used both to propagate updates and to discover other replicas during replica addition and removal. The propagation of updates between replicas is done through the use of flooding along graph edges, exchanging only deltas of data between nodes and not the full replica. The process is divided into two phases. The first phase notifies the occurrence of an update, and the second phase is started by the client that requests the update. Pangaea relies on optimistic replication. Conflict resolution is done with the use of version vectors and the last-writer-wins rule, guaranteeing eventual consistency. This approach does not guarantee that, at any given moment, all replicas are consistent.

Contrarily to the FEW Phone FS, Pangaea is designed for wide-area networks, and does not address the specific problems created by mobile environments. The FEW Phone FS is designed to allow users to access their personal data, even in disconnected machines.

Pangaea propagates updates among replicas through the use of flooding. This forces the system to maintain a strongly connected graph of all replica for the same file. The FEW Phone FS uses a centralized mobile server, that maintains the most recent version of the user's data. As the user usually carries this mobile server, it can be used for synchronizing and for providing access to the most recent version of the user's data in disconnected machines. The use of the Data Source Verification module allows the FEW Phone FS to use other sources for retrieving the users data, thus minimizing communications with the central server, and providing some form of some form of peer-to-peer interaction, as used in Pangaea.

5.1.2 Lookaside Caching

Lookaside caching [30] is a technique that intends to combine distributed file systems (DFS) and portable storage devices (PSD). The main purpose of this system is to take advantage of the

storage capacity and performance of PSD, allowing them to be used for caching accessed files in a DFS, thus improving the availability of those systems. It extends the definition of meta-data to include a cryptographic hash of the data contents. In this way, a client with a valid meta-data for an object, can use the hash to redirect the fetch of data contents. If a mounted PSD has a file with matching length and hash, the client can obtain the contents of the file from that device without the need of accessing the file server. The hash guarantees that the file contents are the same. The DFS cache coherence protocol is responsible to track the client and server state. The system imposes no degradation of consistency since it uses the hash, not only to redirect access to the data contents, but also to verify data consistency.

Lookaside caching's design only allows for PSDs to be used as availability extensions and performance enhancers for DFSs. Unlike the FEW Phone FS, these devices only allow users to access a small portion of their data during disconnection. The FEW Phone FS allows users to always access the full portion of their personal data, independent of machine or network connectivity.

Although Lookaside caching allows for disconnected operations, updates during disconnection need to be manually propagated back into the DFS server, thus forcing the user to explicitly deal with possible consistency and coherency issues. The FEW Phone FS automatically maintains the most recent version of the user's data on the user's mobile phone, using this device to automatically maintain consistency among all replicas. Possible update conflicts are automatically resolved by this process.

The FEW Phone FS also allows the integration of other storage systems as data providers, by using the Data Source Verification module, thus taking advantage of these systems to improve storage capacity and availability.

5.1.3 PersonalRAID

PersonalRAID [28] is a mobile storage system designed to support the synchronization of a collection of disconnected and distributed computers from a single user.

The main purpose of this system is to take advantage of mobile storage devices as the means to propagate updates between disconnected and distributed personal computers from a single user, allowing the synchronization of the user's data. For this purpose, users must always carry with them a mobile storage device. This mobile storage device is a central part of the file system and provides: the illusion of a single name space, allowing users to use a common name space

while accessing different storage devices (local file system + mobile storage device); and also ensuring data reliability, by maintaining data replicas on a number of different storage devices. A major concern of the system is to impose the minimum overhead on file system access as possible.

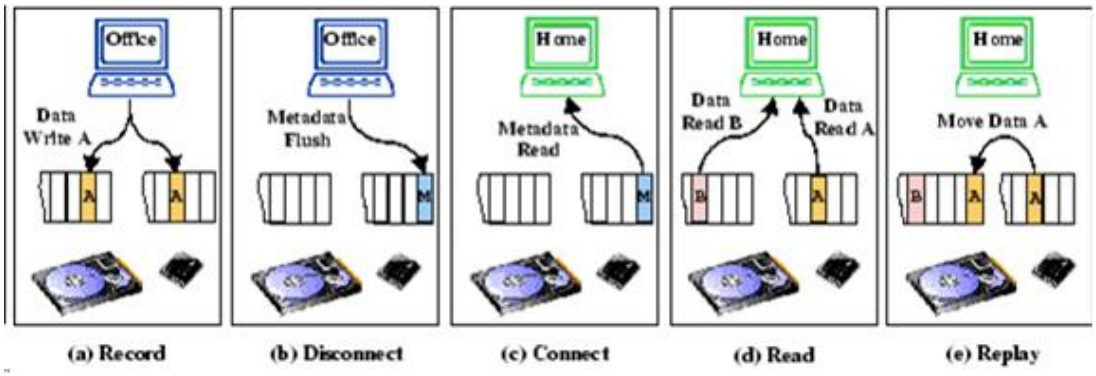

Figure 5.1 PersonalRAID operations as shown in [28].

To accomplish this, PersonalRAID provides several different operations as depicted in Figure 5.1. When a write operation is preformed, data is written to both the local and the mobile storage devices. On disconnect and connect operations, the meta-data is flushed to or read from the mobile storage device respectively. A read operation fetches data from both devices in order to determine which one has the latest version. Finally a replay operation propagates the version on the mobile storage device to the local storage device.

These operations allow the mobile storage device (named Virtual-A) to carry only the updates that have not yet been propagated to all the user's personal computers. When all replicas of a file are consistent, the update on the Virtual-A (VA) is removed and its space is freed. The five operations in conjunction with the invariant that "a copy of any data resides on at least two devices" represent the basis for the system to work properly.

Unlike the FEW Phone FS, PersonalRAID only maintains updates in the PSD. This makes it impossible for a user to access all his data in a new computer. The recent improvements in PSD's capacity allowed us to remove this limitation by carrying most of a user's data in his mobile phone.

The loss of the mobile phone, in the FEW Phone FS does not pose any problem, since all replicas of the data are kept in all machines where the FEW Phone FS has been accessed. The user only needs to "visit" one of the previous computers in order to "restart" the server.

For dealing with the specific problem of storage capacity, the FEW Phone FS integrates a Data Transcoding module and a Data Source Verification module. This allows for the system to reduce the storage usage of the mobile device, while guaranteeing not only access to the full-fidelity version of the files, but also to integrate other remote servers for providing more storage capacity.

5.1.4 Footloose

Footloose [19] is a user centered data store that manages data replication in multiple personal devices, and reconciles conflicts amongst concurrent updates. Footloose ensures that updates are reflected in all devices that "show" interest to them, be it in a direct way on by means of other intermediate devices that "know" about these updates. The main idea of this system is to eliminate the need of a Home PC that is responsible for making sure that the data from the peripheral devices is both persisted and consistent. Due to the growth of mobile devices and the increase of mobility, the authors believe that it will no longer be adequate to expect for consistency to be maintained by one-to-one synchronization directly with a Home PC.

For this, Footloose introduces two main ideas for maintaining the user's data store:

- Physical eventual consistency on a human time-scale.
- Application-based selective conflict resolution.

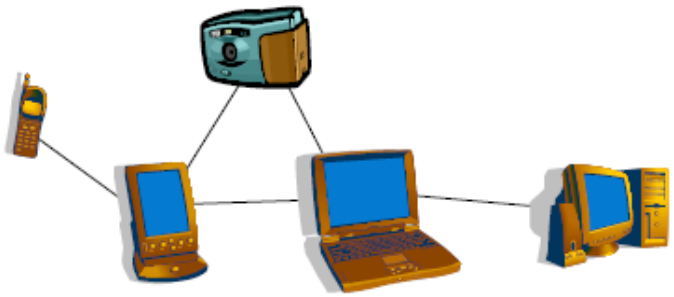

Figure 5.2 Footloose interaction as shown in [19].

Physical eventual consistency is possible since the propagation of updates is based on physical presence of the intervenient devices, as presented in Figure 5.2. One scenario of use is, for

example, when a user stores a new telephone number on a mobile phone's address list. When that mobile phone comes into the presence of that user's PDA (i.e. in the PDA action radius), it exchanges the newly created phone number with that device. At this moment both the PDA and the user mobile phone have the same contact. When the user's PDA is in range of the user's home PC, it can, once again, synchronize the phone number with the contact list stored in that PC. In this example all the devices "know" the data and its "meaning", but some devices may be used only as carriers of information, since they cannot "understand" that information. For example, consider a user that keeps a close check of his money spending in a worksheet in his home PC. If the user makes an online purchase in his work PC, and that PC "knows" that the home PC is interested in that information, it can pass the information to the user's mobile phone. The mobile phone has no way of "knowing" the information it carries, since it cannot interpret it. But the mobile phone has the knowledge that the information is "addressed" to the home PC, so when both devices (i.e. the mobile phone and the home PC) come into wireless range of one another, they exchange that data.

In this context, all devices exchange information with one another in a way that ensures data replication and availability. For this to be possible, each device has the knowledge of which other devices it knows and what data they are interested in. When two devices communicate with each other, they exchange the known updates on replicated data and also on the data which devices are interested on, as exemplified before.

Since Footloose works in a primarily disconnected network environment it uses an optimistic replication approach that allows any user to update data. Eventual consistency guarantees that data replicas converge to the same state given that all update messages are propagated to all participating devices. Since update propagation is executed in an epidemic way between devices, as explained before, the device closest to the user will have the most up-to-date data and it will propagate this information to other devices. For addressing conflict-resolution, Footloose divides devices into two broad classes: smart devices and dumb devices. Smart devices have conflict-resolver applications that are able to resolve conflicting updates. Dumb devices can only store and forward data. Since conflict resolution is application based, dumb devices to some data types can be smart devices for other data types. The applications installed on a device determine the selection of conflicts it can resolve. All devices let all other updates (i.e. the ones that cannot be resolved by that device/application) flow opaquely through the system. Footloose entitles this as selective conflict resolution.

Footloose introduces the concept of physical eventual consistency. This is an important

concept to the design of the FEW Phone FS, allowing for the mobile phone to be used as the means to propagate updates between disconnected nodes. Since the mobile phone will always be carried by the user, it will always keep the most recent version of the user's data.

Unlike FEW Phone FS, Footloose does not take advantage of the availability of high speed network connections for retrieving data from other replicas. For instance, an update to the contact list in the user's work PC would result in the transfer of that update to the user's handheld device. When that device would come into contact range of the user's personal PC, that update is exchanged between the two nodes. The FEW Phone FS would take advantage of the possible existence of a high speed network connection, for retrieving that update from the remote PC, rather than relying on the connectivity with the handheld device. Since, due to communications a mobile device's power consumption is considerable, this minimizes power consumption on that device.

5.1.5 Blue File System

Blue File System [18] (BlueFS) is a distributed file system designed for mobile computing, that allows disconnected operations while reducing energy consumption by integrating portable storage devices. The architecture of this system is based on a centralized server containing the primary replica of each file. Since the major purpose of BlueFS is to minimize energy consumption, it uses a flexible cache hierarchy, which consists of an adaptive cache system, designed to reduce energy usage by efficiently integrating portable storage devices. These devices are used to cache replicas, and enable the system to decide where to access data based upon the performance and energy settings of each available device.

BlueFS was designed with three primary goals:

- Allow the system to automatically monitor and adapt performance and energy characteristics of local, portable, and network storage. BlueFS adapts to storage variability by passively monitoring performance of each local, portable, and network storage option available, and also monitoring each devices power states. On read operations the system sorts the available storage options by estimated performance and energy costs. It then tries to obtain data from the lowest cost device. If that fails, it will try the subsequent devices. On write operations, the BlueFS aggregates updates in a per-device operation logs and asynchronously writes the changes to each device. The asynchronous mode hides the

performance cost of writing to several devices, while aggregation saves energy by amortizing expensive power state transitions across multiple modifications (single long writes vs. many short writes).

- Extend client battery lifetime. This is done by taking into consideration the energy cost in addition to performance when deciding from which device it will read data, by aggregating write operations as described before.

- Transparent support for portable storage devices. BlueFS presents the user with a single system image across all computers and portable storage devices, allowing each portable storage device to act as a persistent file cache. The primary replica of each file resides on the file server, and the client and portable storage devices store second-class replicas that are used to improve performance, reduce energy costs, and support for disconnected operations.

Since this BlueFS allows for disconnected operations and asynchronous updates to improve performance, it supports a weak consistency model. Concurrent update conflicts are automatically detected and flagged for manual or automatic resolution, using a Coda-style conflict resolution protocol.

The FEW Phone FS shares some of the design options of the BlueFS. The BlueFS uses multiple storage devices for maintaining long lived replicas of the access files, using a read from any, write to many strategy. The selection on which device to read from is based on each devices power consumption.

The FEW Phone FS allows for long lived replicas to be created and access in any computer, thus minimizing power consumption since no communication with the server is needed. The FEW Phone FS also takes advantage of the Data Source Verification module to allows for other data source to be used instead of the mobile phone. This has great impact in terms of power consumption and also in terms of performance.

Energy related issues are also dealt by the FEW Phone FS using the Data Transcoding module. The use of this module not only reduces power consumption, but also reduces the mobile phone's storage usage.

5.1.6 EnsemBlue

EnsemBlue [22] is a distributed file system for personal multimedia that incorporates general-purpose computers and consumer electronic devices (CEDs).

EnsemBlue is based on the BlueFS, and it takes advantages of the characteristics of the latter system (described earlier). The main idea of EnsemBlue is to have a central file server that maintains the collection of all multimedia files of all the user's CEDs. Any time a user connects a new CED to any system client (general-purpose computer), the system is responsible for copying the stored files to the server, leaving in this way, the CED with secondary replicas of those files.

In order for EnsemBlue to accomplish this, it adds the following new capabilities to BlueFS for supporting CEDs:

- Persistent queries. A persistent query delivers event notifications to applications that specialize file system behavior. These notifications can be delivered to applications running on any EnsemBlue client, and have low overhead because they reuse the existing cache consistency protocols of the file system to deliver notifications. For example, consider an application that converts 'm4a' music file to 'mp3'. On the first run of the transcoder, it creates a query for all events (existing files, on creation or on update) on files matching the '.m4a' extension. When any file with that extension is detected by the system, the transcoder is notified in order to convert the given file. Persistent queries can also be used for specifying specific files to be cached in specific devices, i.e., the user can define a persistent query that caches all 'mp3' music files into an MP3 player, allowing them to be played while disconnected.

- Namespace Diversity. EnsemBlue supports namespace diversity by creating a common distributed namespace for a user's personal data that is shared among general-purpose computers. The system views files stored on a CED as replicas of files in its distributed namespace. On a CED that mandates a custom organization, EnsemBlue stores data in a local namespace that matches the mandated organization. Changes and modifications made in the EnsemBlue namespace or within CED namespace are automatically propagated to the CED namespace or EnsemBlue namespace, respectively, and shared with other clients.

- Ensemble support. Device ensemble allows multiple mobile computers and CEDs owned

by the same user to self-organize and share data. EnsemBlue allows mobile clients to form ensembles and directly access data from other clients. This is made possible since it presents a consistent view of data within each ensemble, and by propagating changes made on one device to replicas on other ensemble members.

Since EnsemBlue targets multimedia data, it is expected that most files stored in the system will not only be large, but also that reads will dominate writes. In this way the consistency model is designed for a read-mostly workload, using a callback-based cache coherence strategy. These callbacks are used for the server to notify the clients of object modifications. Updates are propagated to the file server asynchronously and conflicts are resolved using a Coda-style conflict resolution protocol.

Since EnsemBlue is built on top of BlueFS, the presented comparison between that system and the FEW Phone FS is also valid here. We will only address the specific characteristics of EnsemBlue, disregarding the inherited ones.

Unlike the FEW Phone FS, EnsemBlue transcodes multimedia data to specific formats based on application needs. This is done by means of what the authors describe as 'persistent queries'. In the FEW Phone FS the Data Transcoding module is used during the reconciliation process for adapting multimedia contents based on the description held by mobile device.

5.1.7 Summary

Table 5.2 presents the comparison of the relevant aspects of the previously described systems and the FEW Phone FS.

As most of the systems, the FEW Phone FS is designed for a mobile environments, and is based on a client-server architecture. Due to power limitations imposed by the use of a mobile device, it also allows for replica updates to be obtained in a peer-to-peer interaction, thus allowing the system to reduce communications between client and server. The reduction in communications between client and server greatly reduces the mobile phone's power consumption. When compared to the most similar system, Footloose, the FEW Phone FS is expected to perform better because it can switch from the client-server interaction to peer-to-peer communications, while Footloose either uses one approach or the other.

The optimistic replication approach in conjunction with an eventual consistency model and automatic conflict resolution, allows for disconnected operations. This not only contributes to the reduction of communications between client and server, but also improves performance

	Systems					
	Pangaea	**Lookaside Caching**	**Personal Raid**	**Footloose**	**BlueFS / EnsemBlue**	**FEW Phone File System**
Architecture	Peer-to-Peer	Client-Server	Client-Server	Client-Server Peer-to-Peer	Client-Server	Client-Server Peer-to-Peer
Replication Model	Optimistic	——	Optimistic	Optimistic	Optimistic	Optimistic
Allows Disconnection	Yes	Yes	Yes	Yes	Yes	Yes
Consistency Model	Eventual Consistency	Strong Consistency	——	Eventual Consistency	Weak Consistency	Eventual Consistency
Conflict Resolution	Automatic Resolution	Manual Resolution	——	Selective Resolution	Coda Style	Automatic Resolution
Environment	Wide Area Environment	Mobile Environment	Mobile Environment	Mobile Environment	Mobile Environment	Mobile Environment

Table 5.1 Comparison between mobile storage systems.

since accessing the file system does not depend on communicating with other nodes. The combination of these characteristics is also used by most of the other system to provide solutions for mobile computing.

5.2 Synchronization and Reconciliation

The following sections present a general overview of some works that are important for defining different approaches related to data synchronization and reconciliation in file systems.

5.2.1 RCS

RCS [29] is a local version control system that allows users to manage file versions and configurations.

RCS' primary function is to manage the evolution of documents. The authors entitle each document version as a revision, and name revision groups to a set of revisions of the same text file. RCS provides flexible selection functions for composing configurations. A configuration is defined as a set of revisions produced by different revision groups, which are selected according to a certain criterion. For example, in order to build a compiler, the correct revisions from the scanner, the parser, the optimizer and the code generator must be combined.

For revision groups to be created, RCS relies on the the check-out check-in model. Checking-in either creates a new revision group or creates a new revision for an existing one. Checking-out obtains the current version of the text file, allowing further updates. For example, suppose a text file f.c. The first check-in will create a new revision group with the content of that file as the initial revision (numbered 1.1). In order for users to edit the file they must check-out the file. When a check-out is preformed, RCS extracts the latest version from the revision group and writes the content to the local file copy (i.e. f.c), enabling users to further edit the file. When a user finishes editing the contents of the file, he will invoke the check-in command, thus creating a new revision (numbered 1.2) for the revision group of that file.

For managing revisions, this system arranges them into a tree. Each tree root is the initial revision of the text document and it has the lowest identification number. The following updates will produce new levels of the tree with consecutive identification numbers. Branches are automatically created when concurrent updates occur.

Due to space concerns, RCS stores revisions as deltas. A classical use of deltas is to store only the differences between revisions. In this way, a document is composed of the merger between all the levels of the tree from the root to the current revision. The problem with this approach is the overhead it imposes. To solve this problem RCS uses reverse deltas. A reverse delta describes how to go backward from the current version to the previous ones. In this ways the latest revision contains the whole document while past revisions contain only the changes according to the latest version.

RCS can prevent concurrent update conflicts by using explicit locking. When locks are used, only one user can modify/edit a revision at any time. Other users may still read it but they have to wait for the lock to be released to be able to modify it. The release of the lock must be done by the user currently holding the lock. For example, when editing a revision in a multi-user environment, a user can explicitly check-out and lock a revision, thus allowing him to edit the revision. When checking-in the new revision, it is the responsibility of the user to releases the lock, thus enabling other users to further edit the revision.

While the approach taken by RCS has proved to be valid for versioning systems, allowing users to access different versions of their documents, it is not an interesting solution for the FEW Phone FS. The FEW Phone FS, as with most file systems, is designed for maintaining only the latest version of each file thus file updates replace previous versions of the file. The approach taken by the FEW Phone FS for handling conflicting updates also reflects this.

5.2.2 Coda

The Coda File System [26] consists of a large collection of clients and a much smaller number of file servers. Each client has a local disk and communicates with the server over a high speed network. The Coda namespace is mapped to individual file servers at the granularity of sub-trees called volumes, and each client dynamically obtains and caches volume mappings. In order to achieve high availability, Coda uses: server replication, allowing volumes to have read-write replicas in more than one server; and allows for disconnected operations, enabling clients to service file requests by relying directly on their cache contents. When disconnection ends, the system propagates modifications back to the server, reverting to server replication.

The set of replication sites for a volume is designated the volume storage group (VSG). The subset of a VSG that is accessible by a client is an accessible VSG (AVSG). Clients only contact their AVSG when file requests cannot be serviced directly by the cache (cache miss). To service a cache miss, Coda nominates one server from that file's AVSG as the preferred server, and obtains the status information and data from it. In parallel it obtains status information from all other members in the AVSG. The system call that caused the cache miss returns successfully only if the collected status from all AVSG members is identical, otherwise the object needs resolution. Validity of cached objects is maintained by callbacks, allowing the server to inform clients whenever a cached file as been modified in the server.

Concurrent updates are detected the first time a file is accessed after two or more partitions reconnect, or when a client propagates the executed updates after being disconnected. When the system detects a version mismatch amongst the replicas while serving a cache miss, the preferred server is informed to resolve the problem. If the resolution is unsuccessful, the system returns an error as the result of the system call that generated the cache miss.

The resolution subsystem is responsible for classifying concurrent updates, propagating benign updates, and preserving evidence from conflicting updates. This is done by maintaining data structures at each server and executing a resolution protocol involving the AVSG of the object being resolved [15].

Every replica of a volume in Coda is associated with a data structure designated by resolution log. This log contains the list of the mutating directory operations of a replica. The resolution protocol uses the log from each replica to deduce and propagate the set of concurrent updates to all replicas. For this purpose, each replica's log is made available to every member of the AVSG. The goal of the resolution protocol is to determine, based on these logs, the set

of concurrent updates missed by each server, and to apply them. Directory conflicts are automatically resolved by the system. File conflicts can be automatically solved using type-specific applications provided by users. If automatic resolution is not possible, files are marked as in conflict, and users must solve conflicts manually before these files can be accessed again.

Directory resolution is preformed entirely on servers, with clients being responsible only for its activation. This is crucial for Coda's goal of scalability, since it eliminates the need for elaborate machinery on servers to keep track of the state of connectivity of other servers. For security reasons resolution cannot be performed by the clients, since it may need to examine or modify regions of the file system for which the client has no access privileges.

Coda performs resolution lazily, since the system only resolves those objects needed to satisfy the triggered system call. This technique minimizes the latency of system calls that trigger resolution and also reduces the peak demands made on servers immediately after recovery from a crash or network partition.

Like Coda, the FEW Phone FS also uses a log structure for maintaining directory updates. Although both system automatically deal with conflicting updates, the solution used by Coda for resolving these conflicts differs from the one used by the FEW Phone FS. While Coda relies on AVSGs for resolving conflicts, the FEW Phone FS automatically resolves directory conflicts during the reconciliation process.

Also, Coda deals with file update conflicts using application specific resolvers. While the solution used in the FEW Phone FS differs from this one, this approach may also be used in the FEW Phone FS.

5.2.3 What is a File Synchronizer?

In [2], the authors present a simple framework for describing the behavior of file synchronizers. Thought the paper is focused on file synchronization, presenting concise definitions of update detections and reconciliation mechanisms, some of the presented issues are more general and relevant to both file and data synchronization.

The presented model requires explicit user invocation, and shows that the synchronization process must be divided into two conceptually distinct phases: update detection, recognizing where updates were made to the individual file system replicas since the last synchronization point; and reconciliation, combining updates to yield the new synchronized state of each replica.

Contrarily to Coda (discussed earlier), the presented method is based on the observation

of the file system state, and not by relying on logged operations. Updates are detected by comparison of the differences between current and the previous state of the file system (i.e. since last synchronization). The information gathered at this stage is then used by a reconciler component that computes the new states of replicas. The reconciler is responsible for, based on the information gathered, determine which operation it should do to each of the replicas in order to achieve a consistent final state.

The paper proceeds in discussing the problems and solutions for different reconciliation scenarios. Here we will just present a simple example. Consider two replicas of the same directory, A and B, that have two different files, x and y. If replica A changes the content of file x to x', and replica B creates a new file z, the result achieved by the synchronization process should produce the same state in both replicas, i.e. both replicas containing files x', y, and z. In this example, some operational issues arise: how does the synchronization process know if file z was created in replica B or if it was deleted from replica A? This can be inferred comparing the differences between current and previous state (although other solutions exist - e.g. tombstones).

The log based solution used by the FEW Phone FS simplifies detecting and solving of directory updates, since there is no need for comparing previous states to the current one. The FEW Phone FS only needs to reproduce the directory updates kept in the log structure.

5.2.4 Summary

Table 5.2 presents a comparison of the relevant aspects the previously described techniques and the solutions used by the FEW Phone FS.

	Systems			
	RCS	**Coda**	**What is a File Synchronizer?**	**FEW Phone File System**
Update Detection	File Versioning	Log Based (directory) Version Vectors (files)	State Comparison	Log Based (directory) Version Vectors (files)
Synchronization	Version Increment	Uses Logged Operations Application Specific Resolvers	State Merging	Replay Logged Operations Latest Version Propagation
Conflict Resolution	Branching	Automatic Resolution	——	Automatic Resolution

Table 5.2 Comparison of synchronization and reconciliation techniques.

File system updates can be divided into two groups: directory and file updates. For detecting

directory updates, the FEW Phone FS uses a log structure, that maintains directory update information, associated with a log version vector. This solution allows the system to detect directory updates simply by comparing each node's log version. The missed operations are then replayed. Coda also handles directory updates using a log. The FEW Phone FS approach of design operations to commute greatly simplifies the resolution solution.

For detecting file updates, the FEW Phone FS relies on version vectors and update flags. To optimize the reconciliation process, when a file is updated, not only the file is marked as updated but also the every directory in the file's path. This allows for entire portions of the file system to be skipped during reconciliation process. The use of version vectors simplifies the reconciliation process since file updates can be detected simply by comparing file versions. They also allow the system to deal with concurrent updates, presenting a simple solution for dealing with them, this approach is also used in Coda. Unlike Coda, the use of update flags prevent the system from adding new entries to version vectors when existing entries can be used instead. This not only allows to improve the system scalability but also reduces the overhead for maintaining these structures.

5.3 Data Management Systems for Mobile Environments

5.3.1 Courier

Courier [13] is a lightweight infrastructure that enables users to access, share and distribute files and URLs across computing systems. It allows users to access the data that reside on their personal computer while they are on the go, and also allows them to bring artifacts they encounter along the way (e.g. data shared by users), back into their personal computers. This is done exploiting the mobile phone as the means to select items to share with or copy from others.

The current version of Courier has been developed with focus on a workspace meeting scenario where the process of sharing data includes: viewing documents as a group; and sharing copies of documents and URLs with the meeting participants.

For this purpose, Courier is responsible for automatically caching document files and URLs accessed by users on their PCs to their mobile phone, allowing these objects to be selected for sharing or displaying. This is done by relying on a meeting room PC host that provides connectivity with the mobile phone through the use of a short-range wireless radio protocol

(like Bluetooth). This meeting room PC also acts as:

- Display controller, allowing the mobile phone to be used for controlling the meeting room shared display, for example, selecting a document for presentation;
- Bridge between mobile phones, enabling the mobile phones to exchange data.

For each of these purposes, the meeting room PC is responsible for connecting to each of the mobile phones present in the meeting room, and to store their data in a shared data pool. Each mobile phone may them browse the shared data, allowing the user to select a document for display, or request a copy of a document to be stored in the mobile phone. These copies are later automatically uploaded and saved to the user's personal computer.

This system offers the participants' limited means to interact with the shared documents, since only users with physical access to the meeting room PC can do so. Also, in order for users to share data with each other, they must always rely on the presence of a third party host responsible for managing the shared data pool.

Unlike the FEW Phone FS, the Courier system is specifically designed for sharing personal data between users. For this purpose, the user must specifically select the information he wants to share. On the other hand, the FEW Phone FS is designed to allows users to access their personal data in any computer independently of network connectivity. Thus, the different goals lead to different solutions.

5.3.2 quFiles

quFiles [32] is a system designed with the purpose of providing specific data management policies for mobile devices. It provides an abstraction, called quFile, designed to encapsulate different versions of a single file, offering multiple representations of a single data object transparently to the user. This allows specific application/device policies to be used when accessing it.

The major purpose of this system is to transparently manage multiple versions of a file, allowing an application or devices to access the version that is most compatible with that device's specifications. For this purpose, the authors define the concept quFile. For example, a quFile can be seen as a collection of different encoding and formats of the same audio file, i.e. the same audio content stored as a 'MP3', 'WAV' and 'M4A' file format with multiple encoding qualities. When a device or application tries to access the quFile, the system is responsible for

returning the version most compatible to that device or application. In order to accomplish this, the system must have the means to transcode data into different formats.

A quFile is implemented as a simple directory that contains different version of the same file. When accessing it, the system can take into account information such as the device's screen/display size, platform, context, connectivity, and battery power, thus delivering the data version most adequate to the device's specifications.

The system offers: transparency, by hiding from the users the existence of a quFile; backward compatibility, since the system offers a transparent to quFiles, there is no need to modify existing application in order to work with them; extensibility, allowing arbitrary resolution policies to be defined for accessing a quFile; and simplicity, by requiring minimum changes to an existing file system to support quFiles.

A major concern of the system is the battery state/power consumption. The current implementation is built on top of the BlueFS (described earlier), and only allows file transcoding to be done in a device that is resource-rich i.e. connected to A/C power and has spare storage space.

The approach taken by quFiles, for transconding files, is different than the one taken by the FEW Phone FS. While quFiles transcodes files when these are stored into the system, using special directories that keep the multiple transcoded versions of each file, the FEW Phone FS transcodes files "on-the-fly" during the first synchronization with the mobile phone, maintaining the full-fidelity version of the file in the computer, thus allowing the system to provide access to the full-fidelity version of the file.

Like quFiles, the transcoding process, in a normal scenario, only happens once for each file (i.e. during the first synchronization with the mobile phone).

6 . Conclusions

This chapter presents the final remarks and possible future work to the FEW Phone File System.

6.1 Final Remarks

The main goal of this work was to design and develop a distributed data management system, that would allow users to access their personal data in any machine, independently of that machine's network connectivity.

For this purpose, the FEW Phone File System was designed to take advantage of current mobile phones' storage and wireless communication capabilities for maintaining the most up-to-date version of the users data, and allowing for replicas to be created and accessed whenever needed.

The use of a mobile device, such as a mobile phone, imposed some limitations on the systems' design that were addressed by the FEW Phone File System, mainly computational, storage and energy limitations.

The optimistic replication approach, taken by the FEW Phone File System, contributes greatly to address these problems, mainly since replicas contained in a host can be accessed without the need for contacting any other nodes, thus helping the system to reduce bandwidth consumption and connectivity requirements.

The combination of the client/server model with peer-to-peer interaction leads to an hybrid architecture. Relying on the mobile phone as a central server allows the system to use it to provide data availability at any time. Relying on peer-to-peer communications allows for power consumption to be reduced and for improving the system's performance, as the presented results have showed.

The Data Transcoding module allows to reduce the multimedia data transferred to the mobile phone, without compromising quality when these files are accessed in the mobile phone. This can be interesting for example, for digital photographs that a user wants to keep in both the mobile phone and his desktop computers.

The Data Source Verification module allows the system to record the sources for the files stored in the system, including remote sources (not in the system). This approach explores the common case where files stored were obtained from the Web or are stored in some remote server

(e.g. CVS/SVN). This module also stores the location of other replicas in the system. This information is used by clients to obtain replica contents from other clients. To our knowledge, the FEW Phone File System is the first system to use this approach to improve performance and availability. Moreover, this also extends battery life by reducing communications with the mobile device, while improving performance.

By combining the Data Source Verification module with the Data Transcoding module, the system allows clients to always access full-fidelity contents. This is a new feature when compared with previous solutions that used data transcoding, since, this way, the systems allows users to access the full-fidelity version of their files.

The results of the performance tests executed, showed that the proposed design imposes minimal overhead to data access on the clients, after initial synchronization. Also, the use of the Data Transcoding and Data Source Verification modules allow the FEW Phone File System to achieve its goals.

6.2 Future Work

During the implementation of the prototype of the FEW Phone File System, some aspects that could improve the system were identified.

Some of the results obtained during the evaluation tests performed on the FEW Phone File System's prototype have shown the existence of some performance aspects that could be improved. The obtained results from multiple file write operations, showed some overhead. This is mainly due to the propagation of update information, for reconciliation use, from children to parent directories, and also due to the contents hash update. This could be addressed by updating the contents hash dynamically on write operations.

In order to further reduce the usage of the communication resources, a data compression algorithm could be used to reduce the data transferred between devices, thus reducing power consumption. The imposed computational overhead, and respective power consumption, must be studied to verify the validity of this improvement.

Also, the Data Source Verification module can be extended in order to be used for other application protocols besides HTTP, and the Data Transcoding module can be further developed for supporting other media types.

An interesting approach could be to further integrate peer-to-peer interaction. This can be

done taking advantage of the source's URLs kept by each replica in order to propagate updates directly amongst replicas in a push model, rather than in a pull model.

An aspect that is important and could have impact is the improvement of the synchronization process, by allowing a node to maintain only a partial replica of a container.

Bibliography

[1] Hiroki Asakawa. Dokan, 2009. `http://dokan-dev.net`.

[2] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer. In *in Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98*, pages 98–108, 1998.

[3] John J. Barton, Shumin Zhai, and Steve B. Cousins. Mobile phones will become the primary personal computing devices. In *WMCSA '06: Proceedings of the Seventh IEEE Workshop on Mobile Computing Systems & Applications*, pages 3–9, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Peter J. Braam. The coda distributed file system. *Linux J.*, page 6, 1998.

[5] Google. Google docs, 2009. `http://docs.google.com/`.

[6] Google. Picasa, 2009. `http://picasa.google.com/`.

[7] R. G. Guy and G. J. Popek. Consistency algorithms for optimistic replication. In *In Proceedings of the First International Conference on Network Protocols. IEEE*, 1993.

[8] Richard G. Guy, John S. Heidemann, Wai Mak, Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus replicated file system. In *In USENIX Conference Proceedings*, pages 63–71, 1990.

[9] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pages 254–265, London, UK, 1999. Springer-Verlag.

[10] John S. Heidemann, Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with ficus. In *in Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, 1992.

[11] Csaba Henk, Miklos Szeredi, Dobrica Pavlinusic, Richard Dawe, and Sebastien Delafond. Filesystem in userspace (FUSE), December 2008. `http://fuse.sourceforge.net/`.

[12] L. B. Huston and P. Honeyman. Disconnected operation for afs. In *MLCS: Mobile & Location-Independent Computing Symposium on Mobile & Location-Independent Computing Symposium*, pages 1–1, Berkeley, CA, USA, 1993. USENIX Association.

[13] Amy Karlson, Greg Smith, Brian Meyers, George Robertson, and Mary Czerwinski. Courier: A collaborative phone-based file exchange system. Technical Report MSR-TR-2008-05, Microsoft Research, May 2008.

[14] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10:3–25, 1992.

[15] Puneet Kumar and M. Satyanarayanan. Log-based directory resolution in the coda file system. In *In Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 202–213, 1993.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[17] Peter Levart. Fuse-j java bindings for fuse (Filesystem in USErspace), 2009. `http://sourceforge.net/projects/fuse-j`.

[18] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In *In Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 363–378, 2004.

[19] J.M. Paluska, D. Saff, T. Yeh, and K. Chen. Footloose: a case for physical eventual consistency and selective conflict resolution. *Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on*, pages 170–179, Oct. 2003.

[20] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, 1983.

[21] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. Nfs version 3 design and implementation. In *In Proceedings of the Summer USENIX Conference*, pages 137–152, 1994.

[22] Daniel Peek and Jason Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation. ACM SIGOPS*, pages 219–232, 2006.

[23] Gerald J. Popek and John S. Heidemann. Replication in ficus distributed file systems. In *in Proceedings of the Workshop on Management of Replicated Data*, pages 5–10. IEEE, 1990.

[24] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, Gerald Popekdepartment, and Computer Science. Resolving file conflicts in the ficus file system. In *In Proceedings of the Summer Usenix Conference*, pages 183–195, 1994.

[25] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *In Proc. OSDI*, pages 15–30, 2002.

[26] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[27] João Soares and N. Preguiça. FEW Phone File System. WSMU'07: Proceedings of the 1st Workshop sobre Sistemas Móveis e Ubíquos, 06 2007.

[28] Sumeet Sobti, Nitin Garg, Chi Zhang, Xiang Yu, Arvind Krishnamurthy R, and Olph Y. Wang. Personalraid: Mobile storage for distributed and disconnected computers. In *In Proc. First Conference on File and Storage Technologies*, pages 159–174, 2002.

[29] Walter F. Tichy and Walter F. Tichy. Rcs - a system for version control. *Software - Practice and Experience*, 15:637–654, 1985.

[30] Niraj Tolia, Michael Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *In Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 227–238, 2004.

[31] Paul Totterman, Vlad Skarzhevskyy, Michael Lifshits, Mina Shokry, Mark Swanson, and Trent Gamblin. Bluecove, 2009. `http://www.bluecove.org`.

[32] Kaushik Veeraraghavan, Edmund B. Nightingale, Jason Flinn, and Brian Noble. qufiles: a unifying abstraction for mobile data management. In *HotMobile '08: Proceedings of the*

9th workshop on Mobile computing systems and applications, pages 65–68, New York, NY, USA, 2008. ACM.